Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.7792

Artigo de Pesquisa/Research Paper

Computação Paralela Usando Arquiteturas ARM: Uma Análise Experimental com Raspberry Pi e Algoritmos de Ordenação de Dados

Parallel Computing Using ARM Architectures: An Experimental Analysis with Raspberry Pi and Data Sorting Algorithms

Lucayan Felipe Teixeira da Silva [Instituto Federal de Rondônia | *lucayanfelips@gmail.com*]
Wanderson Roger Azevedo Dias [Instituto Federal de Rondônia | *wanderson.dias@ifro.edu.br*]

Instituto Federal de Rondônia, Campus Ji-Paraná, R. Rio Amazonas, Jardim dos Migrantes, Ji-Paraná, RO, 76900-730, Brasil.

Resumo. A Computação de Alto Desempenho (*High-Performance Computing* – HPC) combina hardware e software para explorar paralelismo e distribuição de tarefas, sendo essencial para problemas computacionalmente intensivos. Este trabalho apresenta uma análise experimental do desempenho de oito algoritmos de ordenação em arquiteturas ARM (*Advanced RISC Machines*) baseadas em Raspberry Pi, considerando abordagens sequenciais, OpenMP (*Open Multi-Processing*) e MPI (*Message Passing Interface*). Foram avaliados cinco algoritmos clássicos (*Insertion Sort*, *Merge Sort*, *Quick Sort*, *Heap Sort* e *Radix Sort*) e três orientados à execução paralela (*Odd-Even Bubble Sort*, *Bitonic Sort* e *Sample Sort*). Os resultados indicam que o MPI favorece algoritmos divisíveis, como *Merge Sort* e *Sample Sort*, enquanto o OpenMP favorece algoritmos com paralelismo intra-nó, como *Quick Sort* e *Heap Sort*. O *Insertion Sort*, por sua natureza sequencial, não foi avaliado quanto ao *speedup*, enquanto o *Radix Sort* apresentou ganhos limitados, evidenciando a influência do algoritmo, paralelismo e hardware, conforme a Lei de Amdahl.

Abstract. A High-Performance Computing (*HPC*) combines hardware and software to exploit parallelism and task distribution, being essential for computationally intensive problems. This work presents an experimental analysis of the performance of eight sorting algorithms on ARM (*Advanced RISC Machines*) architectures based on Raspberry Pi, considering sequential, OpenMP (*Open Multi-Processing*) and MPI (*Message Passing Interface*) approaches. Five classical algorithms (*Insertion Sort*, *Merge Sort*, *Quick Sort*, *Heap Sort* and *Radix Sort*) and three algorithms designed for parallel execution (*Odd-Even Bubble Sort*, *Bitonic Sort* and *Sample Sort*) were evaluated. The results indicate that MPI favors divisible algorithms, such as *Merge Sort* and *Sample Sort*, while OpenMP favors algorithms with intra-node parallelism, such as *Quick Sort* and *Heap Sort*. Due to its sequential nature, *Insertion Sort* was not evaluated in terms of *speedup*, while *Radix Sort* presented limited gains, highlighting the influence of the algorithm, parallelism and hardware, according to Amdahl's Law.

Palavras-chave: Computação de Alto Desempenho, Algoritmos de Ordenação, Arquitetura ARM, Paralelismo, Raspberry Pi

Keywords: High-Performance Computing, Sorting Algorithms, ARM Architecture, Parallelism, Raspberry Pi

Recebido/Received: 12 March 2026 • **Aceito/Accepted:** 23 August 2026 • **Publicado/Published:** 15 September 2026

1 Introdução

A Computação de Alto Desempenho (*High-Performance Computing* – HPC) tem se consolidado como uma das áreas fundamentais para a resolução de problemas computacionalmente intensivos, especialmente naqueles que requerem grande capacidade de processamento, armazenamento e comunicação eficiente [Góes *et al*., 2023; Hiremath *et al*., 2024]. Ambientes HPC são amplamente utilizados em aplicações críticas como modelagem climática, simulações físicas, bioinformática, inteligência artificial, engenharia, mineração de dados e análise de grandes volumes de dados (*Big Data*) [Rauber and Rünger, 2013].

A evolução da HPC está diretamente relacionada ao avanço das arquiteturas de computadores [Rauber and Rünger, 2013; Parhami, 2006]. Historicamente, o desempenho dos processadores cresceu de forma contínua apoiado pela Lei de Moore, que descreve a duplicação periódica da densidade de transistores [Moore, 1965]. Contudo, esse crescimento deixou de se traduzir em ganhos proporcionais de frequência a partir de meados dos anos 2000, quando o colapso do escalonamento de Dennard impôs um *power wall*: a redução do tamanho dos transistores passou a elevar a densidade de potência e a dissipação térmica sem compensação em desempenho de clock [Esmaeilzadeh *et al*., 2011]. Diante dessas limitações físicas, a indústria migrou para arquiteturas paralelas, inicialmente voltadas a sistemas *multicore* e, posteriormente, a ambientes distribuídos. Assim, o desenvolvimento de aplicações passou a exigir estratégias que aproveitassem o paralelismo de forma eficiente, seja em nível de hardware, por meio de múltiplos núcleos ou *clusters*, seja em nível de software, com o uso de modelos de programação paralela [Parhami, 2006; Andrews, 2001].

Nesse aspecto, processadores ARM (*Advanced RISC Machines*), originalmente desenvolvidos para dispositivos

móveis, passaram a despertar interesse em *data centers* e ambientes de HPC devido ao seu baixo consumo energético e à elevada densidade de núcleos [Ou *et al*., 2012]. Iniciativas como o projeto Mont-Blanc, voltado ao emprego de processadores ARM de 64 bits em sistemas de alto desempenho, reforçam o potencial dessa arquitetura e evidenciam a relevância de *clusters* baseados em computadores de placa única (*Single Board Computers* – SBCs) como plataformas acessíveis para pesquisa em HPC [Ou *et al*., 2012].

Nesse contexto, a exploração eficiente do paralelismo depende também do uso de modelos e bibliotecas de programação paralela capazes de abstrair mecanismos de sincronização, comunicação e distribuição de tarefas. Entre as principais soluções utilizadas destacam-se OpenMP (*Open Multi-Processing*) [OpenMP Architecture Review Board, 2025] e MPI (*Message Passing Interface*) [MPI Forum, 2025], amplamente empregadas na implementação de algoritmos paralelos. O OpenMP é voltado para programação paralela em sistemas com memória compartilhada, permitindo a paralelização de laços e seções críticas de forma relativamente simples [Chandra *et al*., 2000].Por sua vez, o MPI é amplamente reconhecido como o padrão para computação distribuída baseada em troca de mensagens, sendo essencial para a execução de aplicações em *clusters* e supercomputadores [Gropp *et al*., 1994; Pacheco, 2011; Lima *et al*., 2016].

Com o avanço da miniaturização de dispositivos, tornou-se possível criar ambientes computacionais de alto potencial educacional e experimental baseados em arquiteturas ARM, como os *clusters* formados com a plataforma Raspberry Pi. Tais plataformas têm sido amplamente utilizadas em projetos de ensino e pesquisa em Arquitetura de Computadores e Computação Paralela [Lima *et al*., 2016; Ignacio and Dias, 2023], permitindo que conceitos de HPC sejam explorados em ambientes com hardware ARM real, em continuidade com uma tendência mais ampla de adoção de arquiteturas ARM em sistemas de alto desempenho [Simakov *et al*., 2023]. O uso de *clusters* com Raspberry Pi possibilita a criação de ambientes de execução paralela distribuída reais, com suporte nativo a OpenMP e MPI, o que tem ampliado sua adoção em pesquisas experimentais sobre desempenho paralelo em arquiteturas ARM [Ignacio and Dias, 2023].

Dentre os desafios recorrentes da HPC, a ordenação de dados é uma operação fundamental e frequentemente presente em diversas aplicações computacionais. Algoritmos de ordenação são cruciais para o funcionamento eficiente de bancos de dados, sistemas de busca, motores de recomendação, visualização científica e análise de dados em larga escala [Cormen *et al*., 2024]. A escolha do algoritmo e sua estratégia de implementação afetam diretamente o desempenho da aplicação, especialmente quando há restrições de tempo e recursos computacionais.

Assim, o estudo de algoritmos de ordenação de dados é um dos pilares da Ciência da Computação. Obras clássicas como *The Art of Computer Programming*, de Donald Knuth, e as contribuições práticas de Robert Sedgewick fundamentam o entendimento teórico e prático da ordenação [Knuth, 1998; Sedgewick, 1983]. Em contextos modernos, o problema da ordenação em larga escala continua sendo uma das principais preocupações em sistemas de *Big Data* e plataformas como *Apache Spark* e *Google BigQuery* [Zaharia *et al*., 2016; Google Cloud, 2025].

Diferentemente de estudos que avaliam algoritmos de ordenação de dados apenas sob a ótica da complexidade teórica ou do desempenho sequencial, este artigo apresenta o impacto de diferentes modelos de paralelismo (memória compartilhada e memória distribuída) sobre algoritmos de ordenação, considerando tanto a natureza do algoritmo quanto as características arquiteturais do hardware.

Somado a isso, ao utilizar *clusters* homogêneos formados por Raspberry Pi de diferentes gerações (modelos 4 e 5), este estudo permitiu analisar como avanços microarquiteturais, como maior largura de banda de memória, aumento de Instruções por Ciclo (IPC) e maior capacidade de RAM (Random Access Memory), influenciam diretamente a escalabilidade e a eficiência do paralelismo. Dessa forma, este artigo contribui não apenas para a compreensão do comportamento de algoritmos de ordenação de dados em ambientes HPC baseados em arquiteturas ARM embarcadas, mas também para a tomada de decisão sobre o modelo de paralelização mais adequado em função da interação entre algoritmo, modelo de paralelismo e arquitetura de hardware.

Assim, este artigo apresenta uma análise experimental do desempenho de seis algoritmos de ordenação de dados: *Bubble Sort*, *Insertion Sort*, *Heap Sort*, *Quick Sort*, *Merge Sort* e *Radix Sort*. Implementados em três modelos distintos de execução: sequencial, paralelismo com OpenMP (modelo de memória compartilhada) e paralelismo distribuído com MPI (modelo de memória distribuída). Além disso, são incluídos mais dois algoritmos projetados especificamente para execução paralela, como *Bitonic Sort* e *Sample Sort*, amplamente utilizados como referência em estudos de ordenação paralela e ambientes distribuídos. Os experimentos com MPI foram executados em dois *clusters* distintos de baixo custo formados por Raspberry Pi, com o objetivo de avaliar como a arquitetura de execução e a escolha do modelo de paralelismo influenciam o desempenho dos algoritmos em ambientes com restrições de hardware. O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta os trabalhos correlatos; a Seção 3 descreve a metodologia experimental; a Seção 4 apresenta os resultados e discussões; e a Seção 5 apresenta as conclusões e perspectivas para trabalhos futuros.

2 Trabalhos Correlatos

[Ala'anzy *et al*., 2024] analisaram treze algoritmos de ordenação, abrangendo desde métodos básicos, como *Bubble Sort*, até técnicas mais avançadas, como *Bucket Sort*, considerando métricas como tempo de execução, uso de memória, complexidade e paralelizabilidade em diferentes tamanhos de dados (100, 1.000, 10.000 e 100.000 números inteiros).

Os resultados demonstraram que algoritmos de complexidade quadrática apresentaram baixo desempenho para grandes volumes de dados, enquanto algoritmos como *Quick Sort*, *Merge Sort* e *Radix Sort* destacaram-se pela escalabilidade e eficiência. *Heap Sort* e *Shell Sort* apresentaram desempenho consistente em conjuntos de dados de tamanho intermediário, enquanto o *Counting Sort*, apesar de rápido, apresentou elevado custo de memória. Como trabalhos futuros, os autores propõem a ampliação da análise para outros tipos de dados, ambientes paralelos e distribuídos, bem como a investigação

de algoritmos híbridos.

[Yue, 2015] comparou os algoritmos *Bubble Sort*, *Insertion Sort*, *Heap Sort* e *Quick Sort* executados em um computador pessoal e em uma Raspberry Pi modelo B, analisando o tempo de execução para diferentes quantidades de números aleatórios. Utilizando uma arquitetura cliente-servidor, na qual a interface gráfica gerava os dados e o servidor realizava a ordenação, os resultados mostraram que os algoritmos *Bubble Sort* e *Insertion Sort*, ambos de complexidade quadrática, apresentaram desempenho inferior, enquanto *Heap Sort* e, principalmente, *Quick Sort* foram mais eficientes. O estudo confirmou as implicações práticas da complexidade teórica no desempenho real e destacou a relevância dessa análise para dispositivos de baixo consumo energético, como a Raspberry Pi, sugerindo como trabalhos futuros a avaliação de outros algoritmos, tipos de dados e cenários paralelos e distribuídos.

[Rajagopal and Thilakavalli, 2016] compararam algoritmos clássicos de ordenação, incluindo *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Merge Sort*, *Heap Sort* e *Quick Sort*, analisando tanto a complexidade teórica (nos melhores, piores e casos médios) quanto os tempos de execução em diferentes cenários e tamanhos de entrada. Os resultados indicaram que algoritmos quadráticos são inadequados para grandes volumes de dados, embora possam ser úteis em conjuntos reduzidos, enquanto o *Quick Sort* apresentou o melhor desempenho geral, seguido por *Merge Sort* e *Heap Sort*. Como perspectivas futuras, os autores sugerem a inclusão de novos algoritmos, testes com diferentes tipos de dados e a consideração de ambientes distribuídos.

[Aung, 2019] apresentou um estudo comparativo de seis algoritmos clássicos de ordenação: *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*. O trabalho analisou os algoritmos considerando métricas como complexidade temporal, estabilidade e utilização de memória, além de fornecer pseudocódigos e descrições detalhadas das implementações. O autor destacou que algoritmos quadráticos são ineficientes para grandes volumes de dados, apesar de sua simplicidade e adequação para listas pequenas ou parcialmente ordenadas. Já os algoritmos *Merge Sort*, *Quick Sort* e *Heap Sort* apresentaram complexidade $O(n \log n)$, com o *Quick Sort* se destacando pela velocidade média, o *Merge Sort* pela estabilidade e eficiência em listas encadeadas, e o *Heap Sort* pela robustez no pior caso. A análise foi conduzida de forma independente de arquitetura, ampliando sua aplicabilidade teórica.

Além dos algoritmos clássicos, existem algoritmos de ordenação desenvolvidos especificamente para explorar o paralelismo de forma estruturada, especialmente em ambientes distribuídos. Entre esses, destaca-se o *Bitonic Sort*, proposto por [Batcher, 1968], baseado em *sorting networks*, cujo padrão determinístico de comparações o torna particularmente adequado para arquiteturas paralelas, GPUs e sistemas com comunicação previsível. Estudos subsequentes exploraram sua eficiência em ambientes paralelos, demonstrando bom desempenho e escalabilidade quando implementado com múltiplos processadores, apesar de sua complexidade assintótica superior a $O(n \log^2 n)$, especialmente em arquiteturas SIMD, GPUs e sistemas distribuídos [Bilardi and Nicolau, 1989; Peters *et al*., 2010].

Outro algoritmo relevante nesse contexto é o *Sample Sort*, originalmente introduzido por [Frazer and McKellar, 1970], que utiliza técnicas de amostragem para particionar o conjunto de dados de forma balanceada antes da ordenação local. Trabalhos como os de [Blelloch *et al*., 1998] e [Axtmann *et al*., 2017] demonstram que o *Sample Sort* apresenta excelente desempenho em ambientes de memória distribuída, sendo amplamente adotado como referência em implementações paralelas com MPI devido à sua escalabilidade e à redução de desequilíbrios de carga entre processos.

Além da investigação de algoritmos paralelos, estudos recentes também analisaram o impacto da arquitetura de hardware sobre o desempenho de aplicações de HPC. [Kumar *et al*., 2023] compararam clusters ARM (Raspberry Pi 3B) e x86 (Dell OptiPlex 755), ambos com dois nós, executando aplicações MPI para cálculo de π e multiplicação de matrizes. Os resultados mostraram que o cluster x86 apresentou desempenho superior ao ARM em 1,16$\times$ e 2,42$\times$, respectivamente, mostrando que diferenças arquiteturais, como os conjuntos de instruções RISC e CISC, influenciam o desempenho de aplicações intensivas em acesso à memória. Adicionalmente, [Regi *et al*., 2024] destacam que arquiteturas ARM oferecem maior eficiência energética por watt, enquanto sistemas x86 mantêm desempenho superior por núcleo em função das maiores frequências de operação, podendo favorecer aplicações com menor grau de paralelismo.

A partir da análise dos trabalhos correlatos, observa-se que a maioria dos estudos concentra-se na comparação de algoritmos de ordenação de dados sob métricas isoladas, como complexidade temporal ou desempenho sequencial, ou ainda em avaliações restritas a um único modelo de paralelismo. Poucos trabalhos realizam uma comparação direta entre OpenMP e MPI aplicados aos mesmos algoritmos, especialmente em ambientes distribuídos reais baseados em arquiteturas ARM.

Por fim, não foram identificados estudos que avaliem o impacto de diferentes gerações de hardware da plataforma Raspberry Pi sobre o desempenho e a escalabilidade de algoritmos paralelos de ordenação de dados. Dessa forma, esse trabalho contribui para o avanço do estado da arte ao integrar a análise de algoritmos, modelos de paralelismo e arquitetura de hardware em um único estudo experimental, preenchendo lacunas relevantes na literatura.

3 Metodologia para os Experimentos

A metodologia adotada foi estruturada para avaliar experimentalmente o desempenho de oito algoritmos de ordenação de dados: *Bubble Sort (Odd-Even)*, *Insertion Sort*, *Heap Sort*, *Quick Sort*, *Merge Sort*, *Radix Sort*, *Bitonic Sort* e *Sample Sort*. Implementados em três abordagens distintas: execução sequencial, paralelismo em memória compartilhada com OpenMP e paralelismo em memória distribuída com MPI.

O objetivo deste estudo foi compreender como diferentes modelos de paralelização e arquiteturas de hardware influenciam a eficiência computacional. É importante destacar que os resultados de *speedup* e eficiência são analisados individualmente por algoritmo, não sendo realizadas comparações diretas entre algoritmos distintos, em razão das diferenças de carga de trabalho e complexidade computacional.

3.1 Algoritmos de Ordenação de Dados

Os algoritmos de ordenação são essenciais na Ciência da Computação por servirem de base para outras operações fundamentais, como buscas, fusão de dados e compressão. Segundo [Cormen *et al*., 2024], a ordenação pode ser classificada quanto à sua complexidade computacional, estabilidade, uso de memória e adaptabilidade. Assim, neste artigo, foi analisado o desempenho computacional de cinco algoritmos clássicos de ordenação de dados, sendo:

- ***Insertion Sort***: insere elementos na posição correta durante a iteração. Também possui complexidade $\mathcal{O}(n^2)$, mas com melhor desempenho em vetores parcialmente ordenados [Cormen *et al*., 2024];
- ***Merge Sort***: divide e conquista. Complexidade $\mathcal{O}(n \log n)$. Naturalmente paralelizável e eficiente em ambientes distribuídos [Cormen *et al*., 2024];
- ***Quick Sort***: utiliza partição por pivô. Complexidade média $\mathcal{O}(n \log n)$. Paralelizável, porém sensível à escolha do pivô [Cormen *et al*., 2024];
- ***Heap Sort***: baseado em estrutura de heap binária. Complexidade $\mathcal{O}(n \log n)$, mas com alto custo de reorganização interna, o que dificulta paralelização eficiente [Cormen *et al*., 2024];
- ***Radix Sort***: ordenação não comparativa baseada em dígitos. Complexidade $\mathcal{O}(nk)$, onde n é o tamanho da entrada e k o número de operações ou elementos. Rápido sequencialmente, mas com desafios de paralelização [Cormen *et al*., 2024].

Além desses algoritmos clássicos, foram analisados algoritmos projetados especificamente para execução paralela:

- ***Bubble Sort (Odd-Even Transposition Sort)***: variante paralelizável do *Bubble Sort* clássico que alterna fases ímpares e pares, nas quais elementos adjacentes são comparados e trocados simultaneamente, permitindo a execução paralela das comparações independentes [Habermann, 1972; Pacheco, 2011; Lakshmivarahan *et al*., 1984]. Mantém complexidade temporal de pior caso $\mathcal{O}(n^2)$, sendo inadequado para grandes volumes de dados [Cormen *et al*., 2024];
- ***Bitonic Sort***: algoritmo baseado em *sorting networks*, proposto por [Batcher, 1968]. Possui complexidade $\mathcal{O}(n \log^2 n)$ e é amplamente utilizado em arquiteturas paralelas devido ao seu padrão determinístico de comparações, o que facilita sua implementação em ambientes com múltiplos processadores;
- ***Sample Sort***: algoritmo de ordenação paralela baseado em amostragem, no qual um subconjunto dos dados é utilizado para definir divisões balanceadas do conjunto original. Apresenta complexidade média $\mathcal{O}(n \log n)$ e é particularmente eficiente em sistemas distribuídos com memória distribuída, sendo amplamente discutido na literatura de computação paralela [Frazer and McKellar, 1970; Blelloch *et al*., 1998].

Conforme [Knuth, 1998], a escolha do algoritmo ideal depende do contexto da aplicação, do volume de dados, do tipo de paralelismo empregado e da arquitetura subjacente.

3.2 Plataformas de Teste

Os experimentos, tanto na execução sequencial quanto nas abordagens paralelas com OpenMP e MPI, foram realizados em plataformas Raspberry Pi. Os algoritmos implementados utilizando a biblioteca MPI foram executados em dois *clusters* distintos, conforme especificado a seguir:

- **ClusterPi-5**: composto por cinco unidades de Raspberry Pi 5, modelo B, com CPU *quad-core* ARM Cortex-A76 de 2,4 GHz, arquitetura ISA ARMv8-A de 64 *bits* e 8 GB de memória RAM;
- **ClusterPi-4**: composto por cinco unidades de Raspberry Pi 4, modelo B, com CPU *quad-core* ARM Cortex-A72 de 1,5 GHz, arquitetura ISA ARMv8-A de 64 *bits* e 2 GB de memória RAM.

Cada *cluster* é constituído por um nó mestre e quatro nós de processamento. A comunicação entre os nós foi realizada por meio de um *switch Gigabit Ethernet* de 16 portas (modelo TP-Link TL-SG116E), assegurando baixa latência e alta taxa de transferência na rede interna. A ligação das Raspberry Pi's ao *switch* foi realizada por meio de cabos *Ethernet Cat8*, com capacidade nominal de até 40 Gbps. Todos os nós operaram com o sistema operacional Raspberry Pi OS de 64 *bits*.

As Figuras 1, 2 e 3 apresentam, respectivamente, o *setup* geral dos dois *clusters*, a estrutura do ClusterPi-5 e a estrutura do ClusterPi-4.

Figura 1. Setup dos dois clusters.

Figura 2. Estrutura do ClusterPi-5.

A escolha pela plataforma Raspberry Pi foi motivada por três fatores principais. Primeiro, pela coerência metodológica, uma vez que seus processadores ARM Cortex-A estão alinhados ao objeto de investigação deste estudo, em concordância com pesquisas sobre desempenho paralelo em arquiteturas ARM [Lima *et al*., 2016; Ignacio and Dias, 2023]. Segundo, pela eficiência energética, já que processadores ARM apresentam menor consumo de energia que alternativas

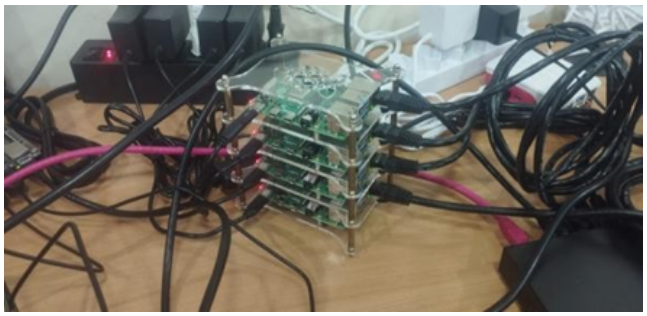

Figura 3. Estrutura do ClusterPi-4.

x86 equivalentes, [Ou *et al*., 2012], característica também observada em sistemas ARM de alto desempenho mais recentes [Simakov *et al*., 2023]. Por fim, pela ampla utilização da plataforma em pesquisas de HPC experimental, favorecendo a comparabilidade dos resultados e a reprodutibilidade dos experimentos [Lima *et al*., 2016].

3.3 Ferramentas de Desenvolvimento e Configuração

Os ambientes de software utilizados nesta pesquisa incluíram:

- Compilador GCC (*GNU Compiler Collection*) para compilação das implementações sequenciais e paralelas baseadas em OpenMP;
- Biblioteca OpenMP para exploração de paralelismo em memória compartilhada;
- Biblioteca MPI (*Message Passing Interface*) para programação paralela em memória distribuída. O MPI é uma especificação de padrão aberto para troca de mensagens entre processos, sendo o MPICH a implementação concreta utilizada neste trabalho para gerenciar a comunicação entre os processos distribuídos [Gropp *et al*., 1994];
- Scripts de automação para compilação e execução dos experimentos, com coleta de resultados e redirecionamento para arquivos de saída, garantindo a reprodutibilidade experimental;
- Todo o código-fonte utilizado nos experimentos encontra-se disponibilizado na seção de declarações complementares apresentada ao final deste artigo, contribuindo para a reprodutibilidade dos resultados obtidos.

3.4 Base de Teste e Justificativa das Cargas de Trabalho

As cargas de trabalho adotadas nos experimentos foram definidas considerando a complexidade computacional dos algoritmos e a viabilidade prática de execução, constituindo uma escolha metodológica. Foram utilizados:

- 1 milhão de números inteiros para os algoritmos de maior complexidade computacional, *Bubble Sort* e *Insertion Sort*, a fim de evitar tempos de execução inviáveis na abordagem sequencial. Essa estratégia de ajuste de carga conforme a complexidade computacional do algoritmo é uma prática metodológica consolidada em estudos de benchmarking paralelo [Hager and Wellein, 2010];
- 100 milhões de números inteiros para os algoritmos *Heap Sort*, *Quick Sort*, *Merge Sort* e *Radix Sort*, permitindo avaliar o comportamento sob cargas elevadas, compatíveis com escalas utilizadas em estudos de referência para algoritmos de complexidade $\mathcal{O}(n \log n)$ em ambientes paralelos [Hager and Wellein, 2010];
- 100 milhões de números inteiros para o algoritmo *Sample Sort*, possibilitando avaliar a eficiência do particionamento por amostragem e o balanceamento de carga nas abordagens paralelas, conforme discutido por [Frazer and McKellar, 1970] e [Axtmann *et al*., 2017] em estudos com cargas de dados de grande escala para esse algoritmo.
- Para o algoritmo *Bitonic Sort*, foi utilizado um vetor de 134.217.728 elementos (2^{27}). Essa escolha decorre de restrições estruturais do algoritmo, que exige que tanto o número de processos quanto o tamanho da entrada sejam potências de dois. Dessa forma, para manter a comparabilidade experimental, adotou-se um tamanho lógico de 100 milhões de elementos, expandido para um tamanho efetivo de 134.217.728 elementos por meio da técnica de *padding*, utilizando valores sentinela que não interferem no resultado final da ordenação. Essa abordagem é consistente com práticas experimentais descritas na literatura para algoritmos baseados em *sorting networks*, nos quais o tamanho da entrada deve ser potência de dois [Batcher, 1968] e [Hager and Wellein, 2010].

Cada experimento foi executado 100 vezes, sendo considerado o tempo médio de execução, a fim de reduzir variações decorrentes de flutuações de carga e latência. Essa prática é amplamente recomendada na literatura de benchmarking de sistemas de alto desempenho, uma vez que medições únicas são suscetíveis a ruídos do sistema operacional, variações de cache e interferências de processos concorrentes [Hager and Wellein, 2010] [Hoefler and Belli, 2015].

Destaca-se que as diferenças nos tamanhos das cargas de trabalho entre os algoritmos são intencionais e decorrem diretamente de suas complexidades computacionais. Algoritmos de complexidade quadrática, como *Bubble Sort* e *Insertion Sort*, tornam-se inviáveis na abordagem sequencial com 100 milhões de elementos, inviabilizando a obtenção da linha de base necessária para o cálculo de *speedup*. Por esse motivo, os resultados não são comparados diretamente entre algoritmos distintos, cada um é avaliado individualmente em relação ao seu próprio tempo sequencial de referência.

3.5 Justificativa do Desenho Experimental

A escolha de três abordagens distintas (sequencial, OpenMP e MPI) teve como base a ampla utilização dessas técnicas em Computação de Alto Desempenho [Chandra *et al*., 2000; Pacheco, 2011].

- Execução sequencial: estabelece a linha de base para comparação;
- OpenMP: explora o paralelismo intra-nó, reduzindo a sobrecarga de comunicação e aproveitando múltiplos núcleos locais;
- MPI: permite a execução distribuída em múltiplos nós, possibilitando ganhos de desempenho quando o custo de comunicação é inferior ao ganho obtido com a divisão do trabalho.

O uso da plataforma Raspberry Pi permitiu investigar o comportamento de algoritmos paralelos em uma arquitetura ARM

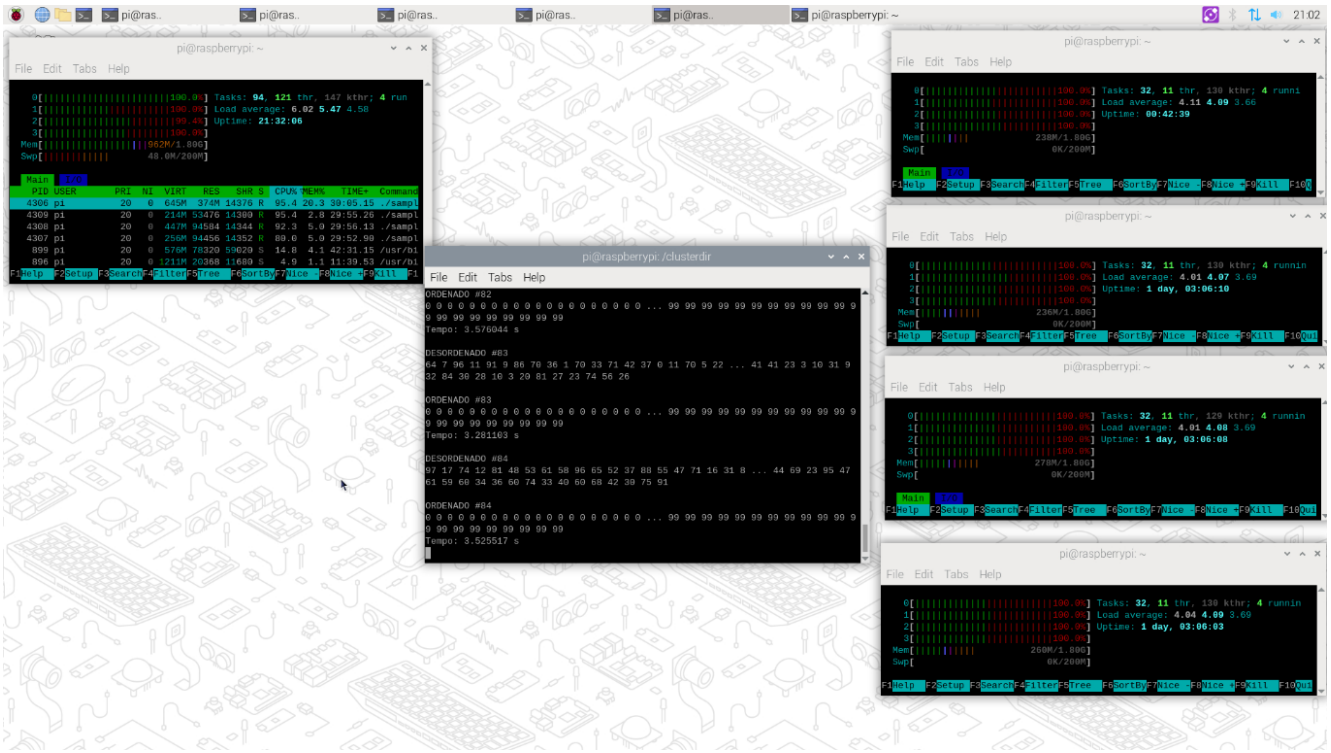

Figura 4. Execução distribuída do algoritmo *Sample Sort* em ambiente MPI no ClusterPi-5, com todos os nós ativos, incluindo o nó mestre, visualizada por meio da ferramenta *htop*.

real e em um ambiente distribuído reproduzível, contribuindo para pesquisas em HPC educacional e para a análise dos impactos arquiteturais no desempenho de aplicações paralelas em sistemas embarcados [Simakov *et al.*, 2023; Rauber and Rünger, 2013].

A escolha por manter o tamanho do problema fixo entre as abordagens está alinhada à perspectiva da Lei de Amdahl [Rauber and Rünger, 2013], que modela os ganhos de paralelização considerando uma carga de trabalho constante. De forma complementar, a Lei de Gustafson [Gustafson, 1988] sugere que, à medida que o tamanho do problema cresce proporcionalmente ao número de processadores, a fração sequencial torna-se relativamente menos limitante, ampliando os ganhos possíveis com a paralelização. Embora este estudo adote cargas fixas, essa perspectiva é relevante para contextualizar os resultados obtidos e orientar trabalhos futuros com cargas escalonáveis.

3.6 Estratégias de Paralelização (OpenMP)

Nas implementações OpenMP, considerou-se o uso de quatro núcleos, correspondentes à arquitetura *quad-core* das plataformas Raspberry Pi. A estratégia de paralelização variou conforme o algoritmo, respeitando dependências de dados e a estrutura de controle de cada implementação.

Algoritmos como *Bubble Sort* e *Insertion Sort* exploraram a paralelização de laços externos, permitindo que diferentes *threads* processassem subconjuntos do vetor com sincronização mínima. No *Quick Sort*, o paralelismo foi obtido por meio da execução concorrente de chamadas recursivas após a etapa de partição.

Em contrapartida, algoritmos como *Heap Sort* e *Radix Sort* apresentaram limitações significativas, devido à necessidade de reorganizações frequentes e à presença de dependências sequenciais, o que restringiu o paralelismo efetivo mesmo com o uso de diretivas OpenMP.

3.7 Estratégias de Paralelização (MPI)

No ambiente MPI, foram utilizados cinco nós da Raspberry Pi em cada *cluster* (ClusterPi-5 e ClusterPi-4), executando quatro processos por nó, totalizando 20 processos MPI. A escolha de quatro processos por nó foi realizada para manter simetria com o número de núcleos disponíveis em cada Raspberry Pi, evitando *oversubscription*. Para o algoritmo *Bitonic Sort*, foram utilizados 16 processos, em função da restrição de potência de dois imposta pelo algoritmo.

Os algoritmos foram adaptados para execução em memória distribuída, com particionamento explícito do vetor de entrada entre os processos. Cada processo realizou a ordenação local de seu subconjunto de dados, utilizando a lógica algorítmica da versão sequencial, com exceção do algoritmo *Insertion Sort*.

A obtenção do vetor global ordenado exigiu etapas de comunicação e fusão dos dados, cujo custo variou conforme o algoritmo. O *Merge Sort* apresentou melhor adequação a esse modelo, devido à sua estratégia natural de divisão e fusão. Em contraste, algoritmos como *Quick Sort* e *Insertion Sort* demandaram maior sincronização e redistribuição de dados, resultando em maior sobrecarga de comunicação e desbalanceamento de carga.

A Figura 4 apresenta uma captura de tela da execução do algoritmo *Sample Sort* em tempo real no ambiente distribuído, evidenciando a utilização simultânea dos nós do *cluster*, incluindo o nó mestre. A visualização foi realizada por meio da ferramenta *htop*, permitindo observar a distribuição dos processos MPI entre os diferentes nós, bem como a utilização intensiva dos núcleos de processamento durante a execução

Tabela 1. Tempo médio de execução dos algoritmos de ordenação de dados com a microarquitetura RPi 5, obtido a partir de 100 execuções por configuração.

Algoritmo	Tamanho Lógico	Tamanho Efetivo	Tempo de Execução (s)			Speedup	
			Sequencial	OpenMP	MPI	OpenMP	MPI
Bubble Sort (*Odd-Even*)	1M	1M	3151,25	932,47	667,47	3,38x	4,72x
Insertion Sort	1M	1M	1476,03	70,18	1000,44	–	–
Merge Sort	100M	100M	36,03	13,10	9,24	2,75x	3,90x
Quick Sort	100M	100M	14,57	4,38	17,73	3,33x	0,82x
Heap Sort	100M	100M	46,06	49,71	45,40	0,93x	1,01x
Radix Sort	100M	100M	2,46	2,62	9,50	0,94x	0,26x
Bitonic Sort	100M	134M	213,20	56,61	18,33	3,77x	11,63x
Sample Sort	100M	100M	23,98	8,59	1,34	2,79x	17,89x

Tabela 2. Eficiência paralela dos algoritmos na microarquitetura RPi 5.

Algoritmos	OpenMP		MPI	
	Unidade	Eficiência (%)	Unidade	Eficiência (%)
Bubble Sort (*Odd-Even*)	4 Threads	84%	20 Processos	24%
Insertion Sort	–	–	–	–
Merge Sort	4 Threads	69%	20 Processos	20%
Quick Sort	4 Threads	83%	20 Processos	4%
Heap Sort	4 Threads	23%	20 Processos	5%
Radix Sort	4 Threads	24%	20 Processos	1%
Bitonic Sort	4 Threads	94%	16 Processos	73%
Sample Sort	4 Threads	70%	20 Processos	89%

experimental.

3.8 Métricas de Avaliação Paralela

O *speedup* é uma métrica clássica em computação paralela que quantifica o ganho de desempenho obtido com a paralelização em relação à execução sequencial, sendo definido como:

$$S(p) = \frac{T_s}{T_p} \tag{1}$$

onde T_s é o tempo de execução sequencial e T_p é o tempo de execução paralela com p unidades de processamento [Rauber and Rünger, 2013; Pacheco, 2011].

Neste trabalho, o *speedup* foi utilizado para quantificar os ganhos de desempenho das implementações paralelas em relação às versões sequenciais. Além do tempo de execução e do *speedup*, a eficiência paralela foi utilizada como métrica complementar. Também consolidada na literatura de computação paralela, a eficiência é definida como a razão entre o *speedup* obtido e o número de unidades de processamento utilizadas:

$$E(p) = \frac{S(p)}{p} \tag{2}$$

onde valores próximos de 1 (ou 100%) indicam maior aproveitamento dos recursos de processamento [Rauber and Rünger, 2013; Pacheco, 2011].

É importante destacar que valores elevados de eficiência paralela não devem ser interpretados como indicativos de escalabilidade ideal do algoritmo, mas como efeitos estruturais da execução paralela, tais como melhor utilização de cache, redução do conjunto de dados processado localmente ou diminuição da carga de trabalho na memória. Dessa forma, essa métrica é utilizada neste trabalho como complemento ao tempo absoluto de execução e ao *speedup*, devendo ser analisada em conjunto com as características dos algoritmos, a fim de evitar interpretações equivocadas sobre ganhos de desempenho.

4 Resultados e Discussão

Os experimentos foram conduzidos conforme descrito na Seção 3, e os resultados são apresentados e discutidos a seguir. As execuções foram realizadas em plataformas Raspberry Pi, modelos 4 e 5, além dos *clusters* ClusterPi-5 e ClusterPi-4, interligados por rede *Gigabit* e executando o mesmo conjunto de testes.

A Tabela 1 apresenta os tempos médios de execução e os valores de *speedup* obtidos na microarquitetura RPi 5. A figura 5 complementa esses dados ao apresentar uma visão comparativa do tempo de execução entre as plataformas RPi 4 e RPi 5 nas três abordagens (sequencial, OpenMP e MPI), separando os algoritmos por grupo de complexidade computacional. O algoritmo *Merge Sort* apresentou ganhos consistentes nas duas abordagens paralelas, com destaque para a implementação em MPI (3,90x contra 2,75x em OpenMP). O algoritmo é naturalmente adaptado à estratégia de "dividir para conquistar" e possui alta independência entre subproblemas na fase de divisão, permitindo que cada processo MPI manipule blocos de dados de forma isolada, concentrando a comunicação principalmente na etapa final de fusão. Esse comportamento está alinhado à Lei de Amdahl [Rauber and Rünger, 2013], uma vez que a fração paralelizável do algoritmo é elevada, resultando em ganhos relevantes mesmo mantendo o tamanho do problema fixo.

Embora o algoritmo *Insertion Sort* tenha sido avaliado nas três abordagens, as versões adotam estratégias estruturalmente distintas. Em todas elas, o vetor é particionado em blo-

Tabela 3. Tempo médio de execução dos algoritmos de ordenação de dados com a microarquitetura RPi 4, obtido a partir de 100 execuções por configuração.

Algoritmo	Tamanho Lógico	Tamanho Efetivo	Tempo de Execução (s)			Speedup	
			Sequencial	OpenMP	MPI	OpenMP	MPI
Bubble Sort (*Odd-Even*)	1M	1M	6482,21	1742,84	1329,35	3,72x	4,88x
Insertion Sort	1M	1M	2520,11	296,82	1895,00	–	–
Merge Sort	100M	100M	66,43	25,13	20,53	2,64x	3,23x
Quick Sort	100M	100M	27,59	8,65	42,49	3,19x	0,65x
Heap Sort	100M	100M	76,11	86,54	129,17	0,88x	0,59x
Radix Sort	100M	100M	5,36	4,55	12,81	1,18x	0,42x
Bitonic Sort	100M	134M	447,59	203,35	34,41	2,20x	13x
Sample Sort	100M	100M	46,99	21,75	3,41	2,16x	13,78x

Tabela 4. Eficiência paralela dos algoritmos na microarquitetura RPi 4.

Algoritmos	OpenMP		MPI	
	Unidade	Eficiência (%)	Unidade	Eficiência (%)
Bubble Sort (*Odd-Even*)	4 Threads	93%	20 Processos	24%
Insertion Sort	–	–	–	–
Merge Sort	4 Threads	66%	20 Processos	16%
Quick Sort	4 Threads	80%	20 Processos	3%
Heap Sort	4 Threads	22%	20 Processos	3%
Radix Sort	4 Threads	30%	20 Processos	2%
Bitonic Sort	4 Threads	55%	16 Processos	81,3%
Sample Sort	4 Threads	54%	20 Processos	68,9%

cos ordenados individualmente por *Insertion Sort*, diferindo na forma como o paralelismo e a fusão são realizados. Na versão sequencial, os blocos são ordenados individualmente e fundidos em árvore binária, de forma totalmente serializada. Na versão OpenMP, a ordenação dos blocos ocorre em paralelo entre threads, porém a etapa de fusão permanece centralizada e sequencial, constituindo um gargalo à medida que o tamanho da entrada aumenta. Na versão MPI, cada processo recebe um subconjunto via MPI_Scatter, ordena-o localmente e participa de uma fusão progressiva em árvore binária por troca de mensagens, estratégia estruturalmente mais próxima da versão sequencial, porém com custos associados à comunicação entre processos.

Essas diferenças tornam as implementações não diretamente comparáveis entre si, inviabilizando o uso de métricas derivadas, como *speedup* e eficiência paralela. Por esse motivo, a análise desse algoritmo foi restrita ao tempo absoluto de execução. Ainda assim, os resultados ilustram um fenômeno relevante: a distribuição da carga computacional reduz o tempo absoluto de execução ao diminuir o volume processado localmente por cada unidade de execução. Entretanto, isso não representa melhoria algorítmica real, pois o comportamento quadrático permanece presente nos blocos locais, enquanto as etapas de fusão introduzem custos de sincronização, comunicação e combinação dos resultados. O *Insertion Sort* atua, portanto, neste estudo como um caso didático de paralelização por decomposição de domínio, evidenciando simultaneamente o potencial e as limitações dessa estratégia quando aplicada a um algoritmo intrinsecamente sequencial.

O algoritmo *Quick Sort* obteve bom desempenho na implementação OpenMP, explorando de forma eficiente a paralelização recursiva em memória compartilhada. No entanto, na versão MPI, o desempenho foi inferior ao da execução sequencial (0,82x). Esse comportamento é explicado principalmente pelo desbalanceamento das partições decorrente da escolha do pivô, aliado à sobrecarga de comunicação entre processos. Em arquiteturas distribuídas, uma má distribuição inicial dos dados pode sobrecarregar alguns nós enquanto outros permanecem ociosos, reduzindo a eficiência global, fenômeno conhecido como *load imbalance* [Colichio *et al*., 2024; Xavier *et al*., 2019].

O algoritmo *Heap Sort*, de complexidade $O(n \log n)$, apresentou comportamento distinto entre as plataformas e abordagens paralelas. Na RPi 5, a implementação OpenMP apresentou leve degradação de desempenho (0,93x), associada aos padrões de acesso à memória impostos pela estrutura hierárquica do heap. Já a implementação MPI obteve *speedup* de 1,01x, caracterizando estagnação, assim, os custos de comunicação compensaram os ganhos obtidos com a divisão da carga, resultando em desempenho praticamente equivalente ao sequencial.

Na RPi 4, a implementação MPI apresentou degradação real (0,59x), sugerindo que as limitações arquiteturais da plataforma ampliaram os custos associados à execução distribuída. Em todos os cenários, entretanto, a causa estrutural permanece a mesma: as operações de *heapify* introduzem dependências sequenciais que dificultam a divisão equilibrada das tarefas, limitando a escalabilidade do algoritmo.

O algoritmo *Radix Sort*, de complexidade linear $O(nk)$, apresentou tempo sequencial de apenas 2,46 s, e a paralelização não trouxe benefícios: a implementação OpenMP manteve desempenho próximo ao sequencial (0,94x), enquanto a versão MPI foi prejudicial (0,26x). Isso evidencia que algoritmos com tempo de execução sequencial intrinsecamente baixo tendem a não se beneficiar da paralelização distribuída, pois a sobrecarga de comunicação e sincronização supera

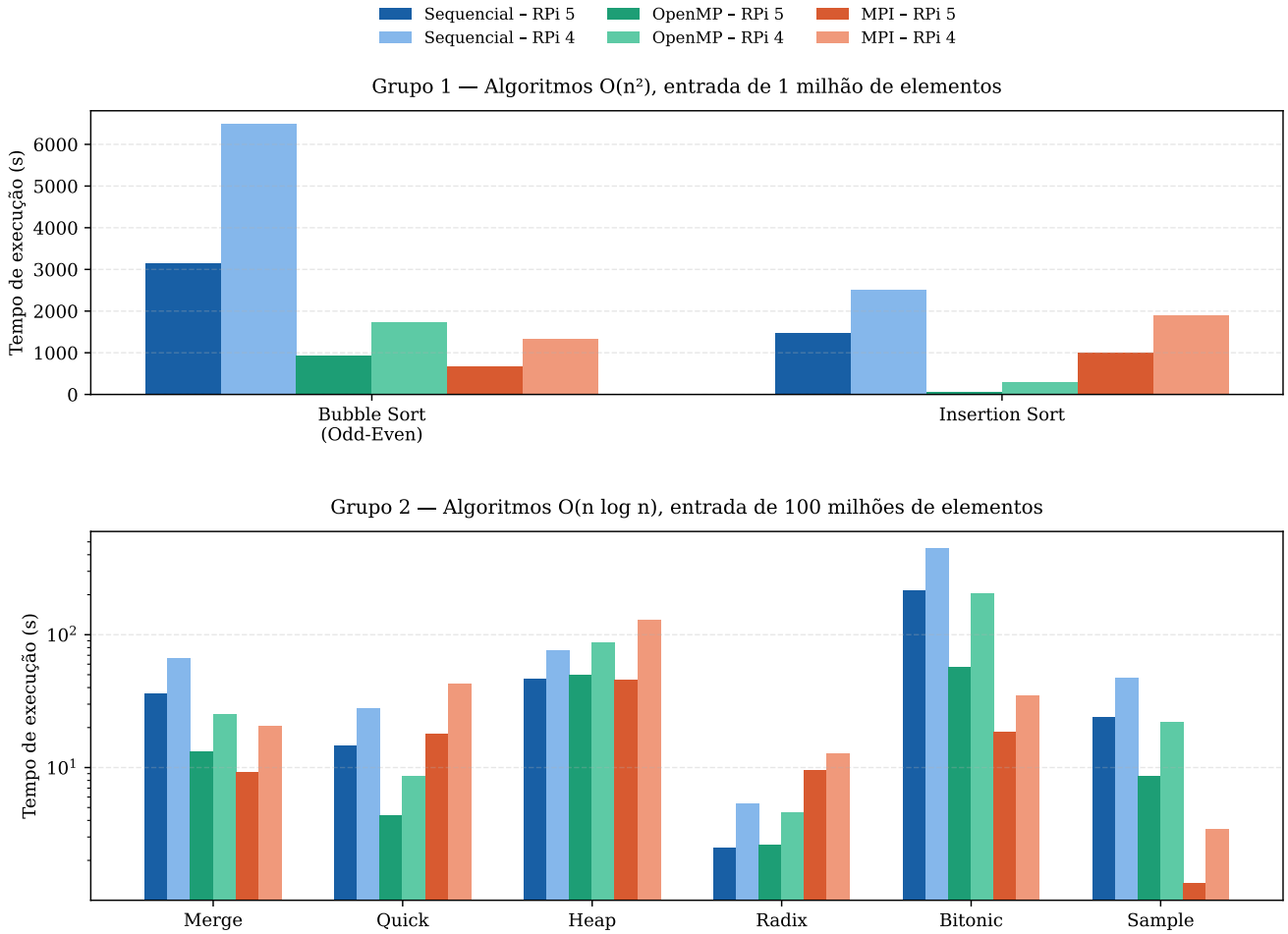

Figura 5. Tempo médio de execução dos algoritmos de ordenação nas plataformas Raspberry Pi 4 e Raspberry Pi 5 utilizando abordagens sequencial, OpenMP e MPI. O Grupo 1 apresenta algoritmos de complexidade $O(n^2)$ com entrada de 1 milhão de elementos. O Grupo 2 apresenta algoritmos de complexidade $O(n \log n)$ com entrada de 100 milhões de elementos.

qualquer ganho potencial.

O *Bitonic Sort* é baseado em uma *sorting network*, na qual a sequência de comparações e trocas é previamente definida. Inicialmente, o vetor é ajustado para o tamanho da próxima potência de dois por meio de *padding*. Em seguida, sucessivas etapas de comparação e troca constroem e mesclam sequências bitônicas até a obtenção da ordenação final. No OpenMP, as comparações de cada etapa são distribuídas entre as *threads*, enquanto no MPI os dados são particionados entre os processos, que realizam trocas periódicas de informações conforme o padrão do algoritmo. Ao final, os resultados são reunidos para compor o vetor ordenado. Essas características favorecem sua execução em ambientes paralelos. Na implementação OpenMP, o algoritmo obteve *speedup* de 3,77x com eficiência paralela de 94% (Tabela 2), resultado associado à regularidade do padrão de comparações, que reduz dependências entre *threads* e favorece a execução em memória compartilhada. O maior ganho foi observado em MPI, com *speedup* de 11,63x e eficiência de 73% utilizando 16 processos. Como a estrutura baseada em *sorting networks* define previamente as etapas de comunicação e comparação, o custo de comunicação torna-se previsível e o desbalanceamento de carga é reduzido, características que favorecem significativamente a execução distribuída.

O *Sample Sort* é executado em quatro etapas. Inicialmente, os dados são divididos entre as unidades de processamento, que realizam a ordenação local de suas respectivas partições. Em seguida, são selecionadas amostras representativas para definir *splitters*, utilizados como limites para a criação de *buckets*. Na terceira etapa, os elementos são redistribuídos de acordo com esses limites. Por fim, cada unidade ordena localmente o *bucket* recebido e os resultados são concatenados. Como os *buckets* já estão globalmente ordenados pelos *splitters*, não é necessária uma etapa adicional de fusão. O *Sample Sort* obteve o melhor desempenho geral entre todos os algoritmos avaliados na microarquitetura RPi 5. Em OpenMP, atingiu *speedup* de 2,79x com eficiência de 70%. Em MPI, alcançou *speedup* de 17,89x e eficiência paralela de 89% com 20 processos, o melhor resultado observado neste estudo. Esse comportamento está associado à estratégia de amostragem empregada pelo algoritmo, que promove uma distribuição mais equilibrada dos dados entre os processos, reduzindo o desbalanceamento de carga e a sobrecarga de comunicação.

O *Bubble Sort* (*Odd-Even*), avaliado com vetor de 1 milhão de elementos em razão de sua complexidade quadrática, apresentou *speedup* de 3,38x em OpenMP e 4,72x em MPI. Esses valores, entretanto, não devem ser interpretados como indicativos de boa escalabilidade algorítmica: seu tempo de execução permanece ordens de grandeza superior ao dos al-

goritmos de complexidade $O(n \log n)$, mesmo após a paralelização. Os ganhos decorrem da decomposição regular do algoritmo e da redução da carga local de cada processo, e não de melhorias na eficiência computacional intrínseca, conforme explica a Lei de Amdahl [Rauber and Rünger, 2013].

A Tabela 3 apresenta os resultados obtidos na microarquitetura RPi 4. Todos os algoritmos apresentaram tempos de execução absolutos maiores quando comparados à RPi 5, reflexo das limitações microarquiteturais do processador Cortex-A72 e da menor capacidade de memória RAM disponível (2 GB contra 8 GB), que pode ter ocasionado uso intensivo de *swap* nas implementações MPI. O impacto é ilustrado pelo *Heap Sort*, cujo tempo em MPI aumentou de 45,4 s no ClusterPi-5 para 129,2 s no ClusterPi-4. O *Merge Sort* manteve boa escalabilidade, com *speedup* superior a 2x em ambas as abordagens paralelas, porém com tempos absolutos cerca de 50% maiores.

O *Quick Sort* apresentou degradação ainda mais severa em MPI na RPi 4 (0,65x), reforçando que o desbalanceamento de carga decorrente da escolha do pivô é agravado pelas limitações arquiteturais da plataforma. O *Radix Sort* manteve o mesmo padrão da RPi 5: bom desempenho sequencial, ganho marginal em OpenMP (1,18x) e forte degradação em MPI (0,42x).

A Tabela 4 mostra que a eficiência paralela foi, em geral, inferior à obtida na RPi 5, especialmente nas implementações MPI. O *Merge Sort* destacou-se novamente como o algoritmo clássico mais robusto frente às restrições da plataforma, mantendo eficiência paralela MPI de 16%, superior à dos demais algoritmos clássicos avaliados. Os algoritmos projetados para execução paralela, *Bitonic Sort* e *Sample Sort*, mantiveram eficiências MPI de 81,3% e 68,9%, respectivamente, demonstrando robustez mesmo sob as limitações da RPi 4.

O *Heap Sort*, devido à sua estrutura hierárquica e às dependências entre etapas de reorganização, apresentou baixa escalabilidade, resultando em estagnação ou degradação dependendo da plataforma. O *Radix Sort*, por sua vez, já apresenta baixo tempo sequencial de execução, fazendo com que o overhead de comunicação e sincronização supere os ganhos potenciais da paralelização.

A comparação entre as plataformas RPi 4 e RPi 5 evidencia que fatores microarquiteturais (como largura de banda de memória, capacidade de RAM e eficiência do processador) são tão relevantes quanto a estratégia de paralelização adotada. Ressalta-se que valores mais elevados de *speedup* ou eficiência eventualmente observados na RPi 4 não indicam melhor desempenho absoluto, mas refletem a influência do tempo sequencial de referência e do *overhead* associado à execução paralela.

5 Conclusões e Trabalhos Futuros

Os resultados obtidos demonstram que não existe um modelo de paralelismo universalmente superior, sendo a escolha fortemente dependente da estrutura do algoritmo, das características do hardware e dos custos associados à comunicação e à sincronização. Algoritmos com alta paralelabilidade e baixo custo de fusão, como o *Merge Sort*, apresentaram melhor desempenho no modelo MPI em ambientes distribuídos. De forma complementar, algoritmos projetados especificamente para execução paralela, como *Bitonic Sort* e *Sample Sort*, demonstraram maior robustez e escalabilidade no modelo MPI, independentemente da microarquitetura utilizada. O *Sample Sort* destacou-se pelo excelente balanceamento de carga proporcionado pela estratégia de amostragem, enquanto o *Bitonic Sort* beneficiou-se de seu padrão determinístico de comunicação, reduzindo os efeitos da sobrecarga e das limitações arquiteturais.

Entre os algoritmos clássicos, o *Merge Sort* destacou-se pelo melhor equilíbrio entre desempenho absoluto, escalabilidade e eficiência paralela em ambientes distribuídos. O *Quick Sort* mostrou-se eficiente no modelo OpenMP, mas inadequado para MPI devido ao desbalanceamento de carga decorrente da escolha do pivô. *Heap Sort* e *Radix Sort* também não se beneficiaram da paralelização, pelas razões estruturais já discutidas na seção 4 como dependências entre etapas de reorganização e baixo tempo sequencial de execução.

A comparação entre o ClusterPi-4 e o ClusterPi-5 evidenciou que, além do número de núcleos e da frequência de *clock*, fatores como largura de banda da memória, capacidade total de RAM e melhorias microarquiteturais são determinantes para o desempenho. A Raspberry Pi 5, equipada com CPU ARM Cortex-A76 e 8 GB de memória RAM, apresentou tempos de execução inferiores em todos os algoritmos avaliados. Esses resultados refletem princípios fundamentais da computação paralela: a Lei de Amdahl [Rauber and Rünger, 2013], que enfatiza a influência da fração sequencial sobre os ganhos obtidos com a paralelização, e, de forma complementar, a Lei de Gustafson [Gustafson, 1988], que sugere que algoritmos com elevada parcela paralelizável tendem a se beneficiar ainda mais à medida que o tamanho do problema e o número de processadores crescem conjuntamente, o que torna relevante a continuidade de estudos em cenários de maior escala.

Como trabalhos futuros, sugere-se: (i) a implementação de abordagens híbridas que combinem MPI e OpenMP, visando otimizar o uso do hardware e reduzir a sobrecarga de comunicação; (ii) a construção de *clusters* heterogêneos compostos por Raspberry Pi 4 e Raspberry Pi 5, possibilitando a investigação de estratégias de balanceamento de carga em ambientes com hardware não uniforme; e (iii) a integração dos algoritmos avaliados a *frameworks* de *Big Data*, como *Apache Spark* ou *Hadoop*, a fim de analisar seu desempenho em sistemas distribuídos de maior escala e (iv) a extensão do estudo experimental para incluir uma comparação direta entre clusters ARM e x86 de configuração equivalente, nos moldes da metodologia empregada por [Kumar *et al.*, 2023], permitindo quantificar o impacto da arquitetura de conjunto de instruções (RISC versus CISC) sobre o desempenho e a escalabilidade dos algoritmos de ordenação avaliados.

Declarações complementares

Agradecimentos

Os autores agradecem ao Instituto Federal de Rondônia (IFRO) pelo suporte institucional que possibilitou a realização deste trabalho.

Financiamento

Esta pesquisa foi financiada pelo Instituto Federal de Rondônia (IFRO), por meio do Programa Institucional de Bolsas de Iniciação em Desenvolvimento Tecnológico e Inovação (PIBITI), conforme Edital n.º 7/2024/REIT–PROPESP/IFRO.

Contribuições dos autores

Lucayan Felipe Teixeira da Silva contribuiu para a implementação dos algoritmos, execução dos experimentos, análise dos resultados e redação do manuscrito. Wanderson Roger Azevedo Dias contribuiu para a concepção do estudo, concepção metodológica, supervisão do trabalho, análise crítica dos resultados e revisão do manuscrito. Ambos os autores leram e aprovaram a versão final do manuscrito.

Conflitos de interesse

Os autores declaram que não têm nenhum conflito de interesses.

Disponibilidade de dados e materiais

Os conjuntos de dados e os códigos-fonte gerados e analisados durante o estudo atual estão disponíveis publicamente no repositório GitHub em `https://github.com/LucayanFelipe/sort_algorithms_article`.

Outras informações relevantes

Não houve necessidade de aprovação por comitê de ética, uma vez que o estudo não envolveu seres humanos nem dados sensíveis. Ferramentas de inteligência artificial generativa foram utilizadas como apoio pontual à revisão linguística, organização textual e auxílio no desenvolvimento de trechos de código, sempre sob supervisão dos autores, sem comprometer a concepção metodológica, a implementação final dos algoritmos, a execução dos experimentos ou a análise dos resultados.

Referências

Ala'anzy, M. A., Mazhit, Z., Ala'anzy, A. F., Algarni, A., Akhmedov, R., and Bauyrzhan, A. (2024). Comparative analysis of sorting algorithms: A review. In *Proceedings of the 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, pages 88–100, Melbourne, Australia. DOI: 10.1109/ISCMI63661.2024.10851593.

Andrews, G. R. (2001). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Boston, MA, 2 edition.

Aung, H. H. (2019). Analysis and comparative of sorting algorithms. *International Journal of Trend in Scientific Research and Development*, 3(5):1049–1053.

Axtmann, M., Witt, S., Ferizovic, D., and Sanders, P. (2017). In-place parallel super scalar samplesort (ips^{4}o). In *Proceedings of the 25th European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*. DOI: 10.4230/LIPIcs.ESA.2017.9.

Batcher, K. E. (1968). Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*. DOI: 10.1145/1468075.1468121.

Bilardi, G. and Nicolau, A. (1989). Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2). DOI: 10.1137/0218014.

Blelloch, G. E., Leiserson, C. E., Maggs, B. M., *et al*. (1998). A comparison of sorting algorithms for the connection machine cm-2. *Theory of Computing Systems*. DOI: 10.1145/113379.113380.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA.

Colichio, H., Pimenta, Y., Borges, P., Menecucci, M., Barros, L., and Guardia, H. (2024). Ordenação paralela com openmp e cuda. In *Escola Regional de Alto Desempenho de São Paulo (ERAD-SP)*, pages 9–12, Rio Claro, SP, Brasil. DOI: 10.5753/eradsp.2024.239919.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2024). *Algoritmos: Teoria e Prática*. GEN LTC, 4 edition.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. DOI: 10.1145/2000064.2000108.

Frazer, W. D. and McKellar, A. C. (1970). Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3). DOI: 10.1145/321592.321600.

Google Cloud (2025). Bigquery: Managed data warehouse for analytics.

Gropp, W., Lusk, E., and Skjellum, A. (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5). DOI: 10.1145/42411.42415.

Góes, L. F. W. *et al*. (2023). Challenges in high-performance computing. *Journal of the Brazilian Computer Society*, 29(1). DOI: 10.5753/jbcs.2023.2219.

Habermann, A. N. (1972). Parallel neighbor sort (or the glory of the induction principle). Technical Report CS-72-112, Carnegie-Mellon University.

Hager, G. and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Boca Raton, FL.

Hiremath, S. K. *et al*. (2024). A systematic comprehensive review and survey on high-performance computing systems approaches in scientific applications. *Journal of Computer Science Engineering and Software Testing*.

Hoefler, T. and Belli, R. (2015). Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*, Austin, TX. DOI: 10.1145/2807591.2807644.

Ignacio, A. L. J. and 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)*, pages 41–48, Porto Alegre, RS, Brasil. DOI: 10.5753/wscad_estendido.2023.235967.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

Kumar, D., Rajput, M. A., Kumar, P., Ahmed, S., Bhatti, M. S., and Tsetse, A. (2023). Performance evaluation of ARM-based versus x86-based processors in high performance computing clusters. *Journal of Independent Studies and Research Computing (JISR-C)*, 21(2). DOI: 10.31645/JISRC.23.21.2.6.

Lakshmivarahan, S., Yang, S. K., and Dhall, S. K. (1984). Parallel sorting algorithms. *Advances in Computers*, 23. DOI: 10.1016/S0065-2458(08)60467-2.

Lima, F. A., Moreno, E. D., and Dias, W. R. A. (2016). Performance analysis of a low cost cluster with parallel applicati-

ons and arm processors. *IEEE Latin America Transactions*, 14(11). DOI: 10.1109/TLA.2016.7795834.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8). DOI: 10.1109/N-SSC.2006.4785860.

MPI Forum (2025). Mpi: A message-passing interface standard version 2.1.

OpenMP Architecture Review Board (2025). The openmp api specification for parallel programming.

Ou, Z., Pang, B., Deng, Y., Nurminen, J. K., Ylä-Jääski, A., and Hui, P. (2012). Energy- and cost-efficiency analysis of arm-based clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*. DOI: 10.1109/CC-Grid.2012.84.

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

Parhami, B. (2006). *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, New York.

Peters, H., Schulz-Hildebrandt, O., and Luttenberger, N. (2010). Fast in-place sorting with cuda based on bitonic sort. In *Parallel Processing and Applied Mathematics. PPAM 2009*. Springer. DOI: 10.1007/978-3-642-14390-8_42.

Rajagopal, D. and Thilakavalli, K. (2016). Different sorting algorithm's comparison based upon the time complexity. *International Journal of u- and e-Service, Science and Technology*, 9(8). DOI: 10.14257/IJUNESST.2016.9.8.24.

Rauber, T. and Rünger, G. (2013). *Parallel Programming for Multicore and Cluster Systems*. Springer, 2 edition.

Regi, G., Kunnappally, J. J., and Yunus, R. (2024). Comparative analysis of energy efficiency between ARM and x86 architectures in mobile devices and its implications for human-computer interaction. *Kristu Jayanti Journal of Computational Sciences*, 4. DOI: 10.59176/kjcs.v4i1.2434.

Sedgewick, R. (1983). *Algorithms*. Addison-Wesley.

Simakov, N. A., DeLeon, R. L., White, J. P., Jones, M. D., and Furlani, T. R. (2023). Are we ready for broader adoption of ARM in the HPC community: Performance and energy efficiency analysis of benchmarks and applications executed on high-end ARM systems. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPCASIAWORKSHOP 2023)*. DOI: 10.1145/3581576.3581618.

Xavier, F., Camargo, E. T., and Duarte Jr., E. (2019). Uma implementação mpi tolerante a falhas do algoritmo paralelo de ordenação quickmerge. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, Campo Grande, MS, Brasil. DOI: 10.5753/wscad.2019.8675.

Yue, Y. (2015). Performance evaluation of sorting algorithms in raspberry pi and personal computer. Master's thesis, University of Vaasa.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2016). Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11). DOI: 10.1145/2934664.