Avaliação de Algoritmos de Ordenação de Dados em Ambiente de HPC com Raspberry Pi

Lucayan Felipe Teixeira da Silva, Wanderson Roger Azevedo Dias

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

{lucayanfelips, wradias}@gmail.com

Abstract. *High-Performance Computing (HPC) integrates hardware and software to exploit parallelism and task distribution, being fundamental in several scientific domains. This article evaluates six sorting algorithms under three approaches: sequential, shared-memory parallelism (OpenMP), and distributed parallelism (MPI), executed in Raspberry Pi platforms. The results show that MPI is more advantageous for naturally divisible algorithms, such as Merge Sort and Bubble Sort, while OpenMP achieves better performance in algorithms with strong intra-node parallelism, such as Insertion Sort and Quick Sort. Algorithms that are already efficient in sequential execution, such as Radix Sort and Heap Sort, showed little benefit from parallelization. The comparison between RPis clusters revealed that architectural advances lead to performance gains, highlighting that the choice of the parallelism model depends both on the algorithm's nature and on the execution architecture.*

Resumo. *A Computação de Alto Desempenho (HPC) integra hardware e software para explorar paralelismo e distribuição de tarefas, sendo fundamental em várias áreas científicas. Este artigo avalia seis algoritmos de ordenação de dados em três abordagens: sequencial, paralelismo em memória compartilhada (OpenMP) e paralelismo distribuído (MPI), executados em Raspberry Pi. Os resultados mostram que o MPI favorece algoritmos naturalmente divisíveis, como Merge Sort e Bubble Sort, enquanto o OpenMP apresenta melhor desempenho em algoritmos com bom paralelismo intra-nó, como Insertion Sort e Quick Sort. Já algoritmos eficientes de forma sequencial, como Radix Sort e Heap Sort, pouco se beneficiaram da paralelização. A comparação entre clusters de RPis evidenciou que avanços arquiteturais, resultam em ganhos, destacando que a escolha do modelo de paralelismo depende tanto da natureza do algoritmo quanto da arquitetura de execução.*

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. 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 & Rünger, 2013).

A evolução da HPC está diretamente relacionada ao avanço das arquiteturas de computadores. A estagnação do aumento da frequência dos processadores devido a limitações físicas, como dissipação térmica e consumo energético, descritas por Moore (1965), levou à adoção de arquiteturas paralelas, inicialmente com sistemas *multicore* e, posteriormente, com ambientes distribuídos. Assim, o desenvolvimento de aplicações passou a exigir estratégias que aproveitem o paralelismo de forma eficiente, seja em nível de *hardware*, por meio de múltiplos núcleos ou *clusters*, ou em nível de *software*, com o uso de modelos de programação paralela (Parhami, 2006; Andrews, 2001).

Nesse cenário, destaca-se a importância de bibliotecas como OpenMP (*Open Multi-Processing*) (OpenMP, 2025) e MPI (*Message Passing Interface*) (MPI, 2025), amplamente utilizadas na implementação de algoritmos paralelos. OpenMP é voltado para programação paralela em sistemas

com memória compartilhada, permitindo paralelização de laços e seções críticas com facilidade (Chandra *et al.*, 2000). Por sua vez, MPI é o padrão mais consolidado para computação distribuída baseada em troca de mensagens, sendo essencial para a execução de aplicações em *clusters* e supercomputadores (Gropp, Lusk & Skjellum, 1994; Pacheco, 2011, Lima, Moreno & Dias, 2016).

Com o avanço da miniaturização de dispositivos, tornou-se possível criar ambientes computacionais de baixo custo e alto potencial educacional e experimental, 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 (Richardson & Wallace, 2013; Ribeiro, Sêmeler & Dias, 2024), permitindo que conceitos de HPC sejam explorados em contextos acessíveis, mas com complexidade real. O uso de *clusters* com Raspberry Pi possibilita a simulação de ambientes de execução paralela realistas, suportando tanto OpenMP quanto MPI, o que amplia ainda mais sua aplicabilidade em pesquisas experimentais, conforme (Ignácio & 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.*, 2012). A escolha do algoritmo e sua estratégia de implementação afetam diretamente o desempenho da aplicação, especialmente quando há restrições de tempo e recursos.

Assim, o estudo de algoritmos de ordenação de dados é um dos pilares da Ciência da Computação. Clássicos como *The Art of Computer Programming* de Donald Knuth (1998) e as contribuições práticas de Robert Sedgewick (1983) fundamentam o entendimento teórico e prático da ordenação. 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).

Neste contexto, este artigo apresenta uma análise experimental do desempenho de seis algoritmos de ordenação de dados, sendo eles: *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). Os experimentos com o modelo em MPI, foram executados em dois *clusters*, 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 alguns trabalhos correlatos, a fim, de relacionar o estado da arte com essa pesquisa; a Seção 3 apresenta a metodologia para os experimentos; a Seção 4 apresenta os resultados e discussões através das análises do desempenho computacional dos algoritmos implementados de forma sequencial e paralela; e a Seção 5 finaliza com as conclusões e ideias para trabalhos futuros.

2. Trabalhos Correlatos

O artigo *Comparative Analysis of Sorting Algorithms: A Review* de Ala'anzy *et al.* (2024) analisou treze algoritmos de ordenação, desde métodos básicos, como *Bubble Sort*, até técnicas avançadas, como *Bucket Sort* e outros, considerando 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 mostraram que algoritmos quadráticos tiveram baixo desempenho em grandes volumes, enquanto algoritmos como *Quick Sort*, *Merge Sort* e *Radix Sort* se destacaram pela escalabilidade e eficiência. Os algoritmos *Heap Sort* e *Shell Sort* foram consistentes em conjuntos médios, e o *Counting Sort*, embora rápido, apresentou alto custo de memória. Para trabalhos futuros, os autores propõem ampliar a análise para outros tipos de dados, ambientes paralelos e distribuídos, além de investigar hibridizações de algoritmos.

O artigo *Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer* (Yue, 2015) comparou os algoritmos *Bubble Sort*, *Insertion Sort*, *Heap Sort* e *Quick Sort* executados em computador pessoal (PC) e na Raspberry Pi modelo B, analisando o tempo de execução para diferentes quantidades de números aleatórios. Utilizando uma arquitetura cliente-servidor, a interface gráfica (cliente) gerou os dados e o servidor realizou a ordenação e retornou os resultados. Os testes mostraram que os algoritmos *Bubble Sort* e *Insertion Sort*, ambos de complexidade quadrática,

tiveram baixo desempenho, enquanto *Heap Sort* e, sobretudo, *Quick Sort* foram mais eficientes, confirmando as implicações práticas da complexidade teórica no desempenho real. O autor destacou a importância dessa análise para dispositivos de baixo consumo, como a Raspberry Pi, e sugeriu como trabalhos futuros a avaliação de outros algoritmos, diferentes tipos de dados e cenários paralelos e distribuídos.

O artigo *Different Sorting Algorithm's Comparison based upon the Time Complexity* (Rajagopal & Thilakavalli, 2016) comparou algoritmos clássicos de ordenação, como *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Merge Sort*, *Heap Sort* e *Quick Sort*, analisando complexidade teórica (no melhor, pior e caso médio) e tempos de execução em diferentes cenários e quantidades de elementos. Os resultados mostraram que algoritmos quadráticos são ineficientes em grandes volumes, mas úteis em conjuntos pequenos, enquanto *Quick Sort* se destacou como o mais rápido, seguido por *Merge Sort* e *Heap Sort*. Para futuros estudos, os autores sugerem incluir novos algoritmos, testar em diferentes tipos de dados e considerar ambientes distribuídos.

O autor Aung (2019) no artigo *Analysis and Comparative of Sorting Algorithms* apresentou um estudo comparativo de seis algoritmos de ordenação clássicos: *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*. O trabalho analisou os algoritmos a partir das métricas de complexidade temporal (melhor, pior e caso médio), estabilidade e utilização de memória, fornecendo pseudocódigos e descrições detalhadas de cada método implementado. O autor destacou que algoritmos quadráticos, como *Bubble*, *Selection* e *Insertion Sort*, são ineficientes para grandes volumes de dados, embora sejam de fácil implementação e adequados 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. Aung (2019) fez uma análise matemática independente da máquina/arquitetura, o que ampliou a aplicabilidade teórica da análise dos algoritmos. Para trabalhos futuros, foram sugeridos manipulações de diferentes tipos de dados, análise da consistência de desempenho em cenários diversos e o impacto da complexidade da implementação.

3. Metodologia para os Experimentos

A metodologia adotada foi estruturada para avaliar experimentalmente o desempenho de seis algoritmos de ordenação – *Bubble Sort*, *Insertion Sort*, *Heap Sort*, *Quick Sort*, *Merge Sort* e *Radix Sort*, implementados em três abordagens distintas: execução sequencial, paralelismo em memória compartilhada com OpenMP e paralelismo em memória distribuída com MPI. O objetivo foi compreender como diferentes modelos de paralelização e arquiteturas de *hardware* influenciam a eficiência de execução.

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.* (2012), 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 seis algoritmos clássicos de ordenação de dados, sendo:

- *Bubble Sort*: Comparações repetidas entre elementos adjacentes, com complexidade $O(n^2)$. Ineficiente para grandes volumes de dados (Cormen *et al.*, 2012);
- *Insertion Sort*: Insere elementos na posição correta durante a iteração. Também $O(n^2)$, mas com melhor desempenho em vetores parcialmente ordenados (Cormen *et al.*, 2012);
- *Merge Sort*: Divide e conquista. Complexidade $O(n \log n)$. Naturalmente paralelizável e eficiente em ambientes distribuídos (Cormen *et al.*, 2012);
- *Quick Sort*: Utiliza partição por pivô. Complexidade $O(n \log n)$ em média. Paralelizável, porém sensível à escolha do pivô (Cormen *et al.*, 2012);
- *Heap Sort*: Baseado em estrutura de *heap* binária. Complexidade $O(n \log n)$, mas com alto custo de reorganização interna, o que dificulta paralelização eficiente (Cormen *et al.*, 2012);
- *Radix Sort*: Ordenação não comparativa baseada em dígitos. Complexidade $O(nk)$. Rápido sequencialmente, mas com desafios de paralelização (Cormen *et al.*, 2012).

Onde: n é o tamanho da entrada e k número de operações ou elementos.

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 (sequencial e paralelo com OpenMP e MPI) foram executados em plataformas Raspberry Pi, sendo que os algoritmos implementados usando a biblioteca MPI, executaram em dois *clusters* distintos, conforme especificados a seguir:

- *ClusterPi-5*: 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 RAM;
- *ClusterPi-4*: 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 RAM.

Cada *cluster* possui um nó mestre e quatro nós de processamento. A comunicação entre os nós foi realizada por meio de um *switch Gigabit Ethernet* (modelo TP-Link TL-SG116E) de 16 portas, assegurando baixa latência e alta taxa de transferência na rede interna, e a ligação das RPis com o *switch* foi usando cabo de rede *Ethernet* Cat8 de 40 Gbps. Todos os nós operaram com o sistema operacional Raspberry Pi OS 64 *bits*. As Figuras 1, 2 e 3 apresentam a estrutura dos ambientes experimentais dos dois *clusters*, do *ClusterPi-5* e do *ClusterPi-4*, respectivamente.

Figura 1. Setup dos dois *clusters*

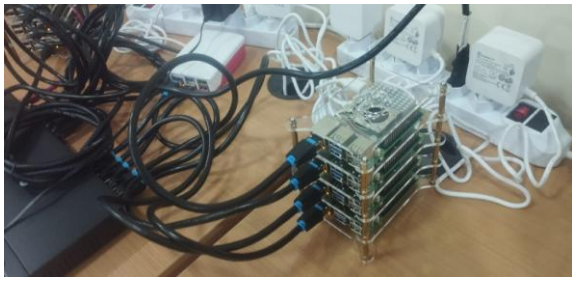

Figura 2. Estrutura do *ClusterPi-5*

Figura 3. Estrutura do *ClusterPi-4*

A escolha pela plataforma Raspberry Pi foi motivada por seu baixo custo financeiro, baixo consumo energético e suporte a arquiteturas *multicore*, permitindo a criação de um ambiente realista para experimentação em HPC, ainda que com recursos limitados (Richardson & Wallace, 2013).

3.3. Ferramentas de Desenvolvimento e Configuração

O ambiente de *software* usado nesta pesquisa, incluiu:

- Compilador GCC (*GNU Compiler Collection*) para as implementações sequenciais e com OpenMP;
- MPICH como implementação do padrão MPI (Gropp, Lusk & Skjellum, 1994), permitindo comunicação entre processos distribuídos;
- Bibliotecas OpenMP e MPI devidamente configuradas para exploração de paralelismo de memória compartilhada e distribuída;

- *Scripts* de automação para compilação e execução, com coleta de resultados e redirecionamento para arquivos de saída, garantindo reprodutibilidade experimental.

3.4. Base de Teste dos Experimentos

Foram definidos dois tamanhos de entrada para os testes, com geração de números inteiros de forma randômica, sendo:

- 1 milhão de números inteiros para ordenação nos algoritmos de maior complexidade computacional (*Bubble Sort* e *Insertion Sort*), visando evitar tempos de execução inviáveis na abordagem sequencial;
- 100 milhões de números inteiros para ordenação nos demais algoritmos (*Heap Sort*, *Quick Sort*, *Merge Sort* e *Radix Sort*), permitindo avaliar o comportamento sob carga elevada.

Cada experimento foi repetido 100 vezes, registrando-se o tempo médio de execução, a fim de reduzir variações decorrentes de flutuações de carga e latência.

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 HPC, conforme (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 ganho de desempenho quando o custo de comunicação é menor que o ganho com divisão de trabalho.

O foco no uso da plataforma Raspberry Pi, visou explorar a eficiência de algoritmos em sistemas com restrições reais de *hardware*, fornecendo subsídios para pesquisas educacionais e prototipagem de soluções HPC em pequena escala (Rauber & Rünger, 2013).

4. Resultados e Discussão

Os experimentos realizados permitiram observar de forma clara como diferentes algoritmos de ordenação de dados respondem a distintos modelos de paralelismo e arquiteturas de *hardware*. As execuções foram realizadas nas RPi's, versões 4 e 5, além do *ClusterPi-5* e *ClusterPi-4*, ambos compostos por cinco nós (um mestre e quatro de processamento), interligados por rede *Gigabit* e executando o mesmo conjunto de testes. O *ClusterPi-5* possui componentes de *hardware* superior comparado com a especificação do *ClusterPi-4* (ver Subseção 3.2). Essa diferença impactou nos resultados, não apenas pela maior frequência de *clock*, mas também pelas melhorias microarquiteturais do processador Cortex-A76, que proporciona execução fora de ordem mais eficiente, maior número de portas de execução e melhor predição de desvios, aumentando assim o IPC (*Instructions Per Cycle*). Além disso, a quantidade maior de memória RAM na plataforma RPi 5, reduziu a probabilidade de utilização de *swap*, especialmente em cargas de testes maiores, o que é crítico em algoritmos que manipulam grandes volumes de dados.

Nos testes realizados na RPi 5 (sequencial e OpenMP) e no *ClusterPi-5* (MPI), o algoritmo *Bubble Sort*, de complexidade $O(n^2)$, confirmou sua ineficiência em execução sequencial, levando quase uma hora para processar 1 milhão de elementos (média de 3.151 segundos). Apesar disso, a versão paralela com OpenMP obteve um ganho de 3,37 vezes, e a versão distribuída com MPI atingiu um *speedup* de 4,72x (ver Tabela 1). Esse ganho extremo não se deve a uma eficiência intrínseca do algoritmo, mas sim ao fato de que, dividindo o vetor entre vários nós, a carga local de processamento foi drasticamente reduzida e a sobrecarga de comunicação foi irrelevante em comparação ao tempo de computação. Este é um exemplo clássico de cenário em que a *Lei de Amdahl* favorece fortemente a paralelização, pois a fração paralelizável é altíssima e o custo de sincronização é mínimo.

Tabela 1. Execução dos algoritmos de ordenação de dados com a microarquitetura RPi 5

Algoritmos	Técnica de Implementação			*Speedup*	
	Sequencial (s)	**OpenMP (s)**	**MPI (s)**	**OpenMP**	**MPI**
Bubble Sort	3151,25	932,47	667,47	**3,37x**	**4,72x**
Insert Sort	1476,03	70,18	1000,44	**21,03x**	**1,48x**
Merge Sort	36,03	13,10	9,24	**2,75x**	**3,90x**
Quick Sort	14,57	4,38	17,73	**3,33x**	**0,82x**
Heap Sort	46,06	49,71	45,40	**0,93x**	**1,01x**
Radix Sort	2,46	2,62	9,50	**0,94x**	**0,26x**

O algoritmo *Insertion Sort*, também $O(n^2)$, obteve resultados distintos, ou seja, o uso de OpenMP gerou um *speedup* elevado (≈21x), mas a versão em MPI apresentou ganho modesto (≈1,48x) (ver Tabela 1). Isso ocorre porque, embora o algoritmo se beneficie do processamento paralelo dentro de um mesmo nó, ele exige inserções ordenadas e fusões frequentes, operações que no contexto de MPI implicam comunicação intensiva e alta sincronização, reduzindo a escalabilidade. Esse comportamento confirma que algoritmos com alta dependência de resultados intermediários são mais adequados a ambientes de paralelismo em memória compartilhada do que à execução distribuída.

O algoritmo *Merge Sort* apresentou comportamento oposto, com ganhos consistentes nas duas abordagens paralelas, mas maior destaque para a implementação em MPI (3,90x contra 2,75x da versão em OpenMP) (ver Tabela 1). O algoritmo é naturalmente adaptado à estratégia de "dividir para conquistar" e possui alta independência entre subproblemas na fase de divisão, permitindo que cada nó MPI processe blocos de forma isolada, com baixo custo de comunicação antes da etapa final de fusão. Isso está alinhado à *Lei de Gustafson*, pois o aumento da quantidade de elementos processados tende a manter o ganho de velocidade proporcional à ampliação do número de processadores.

É possível observar que o algoritmo *Quick Sort* obteve desempenho maior que 3 vezes implementado em OpenMP (≈3,33x), explorando bem a paralelização recursiva. No entanto, na versão em MPI o desempenho foi inferior que na execução sequencial (0,82x). A explicação está no desbalanceamento das partições e na sobrecarga de comunicação durante a fusão, além de possíveis divergências na escolha do pivô. Em arquiteturas distribuídas, a má distribuição inicial dos dados pode causar sobrecarga em alguns nós, deixando outros ociosos, o que reduz a eficiência global, um fenômeno conhecido como *load imbalance*, (Colichio *et al.*, 2024; Xavier *et al.*, 2019).

Já o algoritmo *Heap Sort*, de complexidade $O(n \log n)$, mostrou-se pouco responsivo a paralelização, tanto com OpenMP (0,93x) quanto com MPI (1,01x) (ver Tabela 1). Essa limitação é justificada pelo acesso aleatório frequente e pela estrutura hierárquica do *heap*, que exige reorganizações constantes e dificulta a divisão equilibrada das tarefas. Mesmo com a divisão do vetor, a necessidade de múltiplas operações de *heapify* serializadas impede que o ganho esperado se concretize, reforçando que nem todos os algoritmos de ordenação apresentam escalabilidade relevante.

Por fim, o algoritmo *Radix Sort*, de complexidade linear $O(nk)$, já é extremamente rápido em execução sequencial (2,46s) e, por isso, a paralelização não trouxe benefícios. A implementação com OpenMP manteve desempenho próximo ao sequencial (0,94x), e a versão implementada em MPI foi prejudicial (0,26x) (ver Tabela 1). Isso evidencia que algoritmos com tempo de execução sequencial baixo por natureza, tendem a não se beneficiar da paralelização distribuída, pois a sobrecarga de comunicação e sincronização pode superar qualquer ganho potencial. Esse comportamento também reforça que nem sempre "mais processadores" significa "maior desempenho".

Analisando os testes executados na RPi 4 e no *ClusterPi-4*, observamos que todos os algoritmos apresentaram tempos de execução absolutos maiores, comparados com microarquitetura da RPi 5. O impacto foi mais evidente nas implementações com MPI, principalmente devido à limitação de memória, que pode ter causado uso intensivo de *swap*, degradando o desempenho. Por exemplo, o algoritmo *Heap Sort* em MPI aumentou de 45,4s (*ClusterPi-5*) para 129,1s (*ClusterPi-4*). Já o algoritmo *Merge Sort*, executado na RPi 4, manteve boa escalabilidade, com ganhos de *speedup's* semelhantes aos da microarquitetura RPi 5, ou seja, maior que 2x, com tempos absolutos cerca de 50% maiores. Na execução do algoritmo *Quick Sort*, a degradação da implementação em MPI foi ainda mais severa, refletindo que a microarquitetura da RPi 4 não consegue lidar de forma eficiente com o

volume de comunicação exigido. O algoritmo *Radix Sort* manteve o mesmo padrão observado na RPi 5, ou seja, um bom desempenho sequencial, pequena vantagem ou prejuízo na versão em OpenMP e forte degradação na versão em MPI (ver Tabela 2).

Tabela 2. Execução dos algoritmos de ordenação de dados com a microarquitetura RPi 4

Algoritmos	Técnica de Implementação			*Speedup*	
	Sequencial (s)	OpenMP (s)	MPI (s)	OpenMP	MPI
Bubble Sort	6482,21	1742,84	1329,35	**3,71x**	**4,87x**
Insert Sort	2520,11	296,82	1895,00	**8,49x**	**1,32x**
Merge Sort	66,43	25,13	20,53	**2,64x**	**3,23x**
Quick Sort	27,59	8,65	42,49	**3,19x**	**0,65x**
Heap Sort	76,11	86,54	129,17	**0,88x**	**0,59x**
Radix Sort	5,36	4,55	12,81	**1,18x**	**0,42x**

Os resultados mostram que a escolha do modelo de paralelismo deve considerar tanto a estrutura do algoritmo quanto as características do *hardware*. Algoritmos naturalmente divisíveis, como *Merge Sort* e até mesmo o *Bubble Sort* sob alta carga, tiveram melhor desempenho com MPI, enquanto *Insertion Sort* e *Quick Sort* foram mais eficientes com OpenMP. Já algoritmos com forte dependência entre etapas ou baixa complexidade, como *Radix Sort* e *Heap Sort*, apresentaram pouca ou nenhuma melhoria com o paralelismo. Além disso, a comparação entre a RPi 4 e 5 evidenciou que avanços em arquitetura, largura de banda e memória RAM impactam no desempenho, sendo tão importantes quanto a estratégia de paralelização, especialmente em ambientes de HPC de baixo custo. *Link* dos códigos no *GitHub*: https://github.com/LucayanFelipe/sort_algorithms_article.git.

5. Conclusões e Trabalhos Futuros

Os resultados demonstram que a escolha do modelo de paralelismo ideal depende da estrutura do algoritmo de ordenação, das características do *hardware* e dos custos de comunicação e sincronização. Algoritmos com alta paralelizabilidade e baixo custo de fusão, como o *Merge Sort*, obtiveram melhor desempenho com MPI em ambientes distribuídos, enquanto algoritmos com paralelismo em nível de *thread*, como *Insertion Sort* e *Quick Sort*, se beneficiaram mais do OpenMP e da memória compartilhada. Por outro lado, algoritmos com forte dependência entre etapas ou já eficientes de forma sequencial, como *Radix Sort* e *Heap Sort*, não apresentaram ganhos significativos com paralelização e, em alguns casos, foram prejudicados pelo alto custo de comunicação. Algoritmos ineficientes como o *Bubble Sort* podem mostrar grandes acelerações com MPI quando o custo de comunicação é baixo.

A comparação entre o *ClusterPi-4* e o *ClusterPi-5* mostrou que, além do número de núcleos e frequência de *clock*, fatores como largura de banda da memória, total de RAM e melhorias microarquiteturais são cruciais para o desempenho. A RPi 5, com CPU ARM Cortex-A76 e 8GB de RAM, obteve tempos de execução menores em todos os algoritmos. Esses resultados refletem os princípios da computação paralela, como a *Lei de Amdahl*, que enfatiza a redução da parte sequencial do código, e a *Lei de Gustafson*, que mostra os benefícios da escalabilidade com o aumento do problema. Algoritmos como *Merge Sort* aproveitam bem essas leis, enquanto outros, como *Radix Sort*, rapidamente atingem o limite da escalabilidade.

Para trabalhos futuros, sugere-se: (i) explorar algoritmos paralelos modernos, como *Bitonic Sort*, *Odd-Even Sort* e *Sample Sort*, utilizados em HPC e adaptáveis tanto a memória compartilhada quanto distribuída; (ii) implementar abordagens híbridas combinando MPI e OpenMP, visando otimizar o uso do *hardware* e reduzir a sobrecarga de comunicação; (iii) criar *clusters* heterogêneos com Raspberry Pi 4 e 5, permitindo investigar estratégias de balanceamento de carga em ambientes com *hardware* não uniforme; (iv) integrar os algoritmos a *frameworks* de *Big Data*, como *Apache Spark* ou *Hadoop*, para avaliar seu desempenho em sistemas distribuídos.

Agradecimentos

Os autores agradecem ao Instituto Federal de Rondônia (IFRO), pelo apoio financeiro concedido através do Edital Nº 7/2024/REIT - PROPESP/IFRO - PIBITI Ciclo 2024-2026.

Referências

Ala'anzy, M. A.; Mazhit, Z.; Ala'anzy, A. F.; Algarni, A.; Akhmedov, R.; Bauyrzhan, A. (2024) "Comparative Analysis of Sorting Algorithms: A Review". *In 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, Melbourne, Australia, pp. 88-100.

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

Aung, H. H. (2019) "Analysis and Comparative of Sorting Algorithms". *In International Journal of Trend in Scientific Research and Development (IJTSRD)*, 3(5):1049-1053, August.

Chandra, R.; Dagum, L.; Kohr, D.; Maydan, D.; McDonald, J.; Menon, R. (2000) "Parallel Programming in OpenMP". San Francisco, CA, USA: Morgan Kaufmann, 1st edition, 240p.

Colichio, H.; Pimenta, Y.; Borges, P.; Menecucci, M.; Barros, L.; Guardia, H. (2024) "Ordenação Paralela com OpenMP e CUDA". *In Escola Regional de Alto Desempenho de São Paulo (ERAD-SP)*, Rio Claro, SP, Brasil, pp. 9-12.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2012) "Algoritmos: Teoria e Prática". GEN LTC, 3ª edição, 944p.

Google Cloud. (2025) "BigQuery: Managed Data Warehouse for Analytics", disponível em https://cloud.google.com/bigquery. Acessado em 03/06/2025.

Gropp, W.; Lusk, E.; Skjellum, A. (1994) "Using MPI: Portable Parallel Programming with the Message Passing Interface". Cambridge, Massachusetts, EUA: MIT Press, 1st edition, 307p.

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.

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

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.

Moore, G. E. (1965). "Cramming More Components onto Integrated Circuits". *In Electronics*, 38(8):114-117.

Mpi, (2025) "A Message-Passing Interface Standard Version 2.1", disponível em www.mpi-forum.org/docs/mpi21-report.pdf. Acessado em 10/07/2025.

OpenMP, (2025) "The OpenMP API Specification for Parallel Programming", disponível em http://openmp.org/. Acessado em 12/07/2025.

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.

Rajagopal, D.; Thilakavalli, K. (2016) "Different Sorting Algorithm's Comparison based Upon the Time Complexity". *In International Journal of u- and e-Service, Science and Technology*, 9(8):287-296, November.

Rauber, T.; Rünger, G. (2013) "Parallel Programming for Multicore Cluster Systems". Springer, 2nd ed., 516p.

Ribeiro, G. O.; Sêmeler, L. F.; Dias, W. R. A. (2024) "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)*, São Carlos, SP, Brasil, pp. 65-72.

Richardson, M.; Wallace, S. (2013) "Primeiros Passos com o Raspberry Pi". São Paulo, Brasil: Novatec, 1ª edição, 192p.

Sedgewick, R. (1983) "Algorithms". Addison-Wesley. 1st edition, 552p.

Xavier, F.; Camargo, E. T.; 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, pp. 276-287.

Yue, Y. (2015) "Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer". *Master´s thesis for the degree of Master of Science in Technology*, University of Vaasa - Faculty of Technology, 6 August, 78p.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; & Stoica, I. (2016) "Apache Spark: A Unified Engine for Big Data Processing". *In Communications of the ACM*, 59(11):56-65, November.