



UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO - ICOMP
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO



SEBASTIÃO BICHARRA NETO

PROGRAMAÇÃO PARALELA APLICADA AO PROCESSAMENTO DE IMAGENS E
COMPUTAÇÃO DE ALTO DESEMPENHO: UMA ABORDAGEM EXPERIMENTAL

MANAUS
2025

SEBASTIÃO BICHARRA NETO

PROGRAMAÇÃO PARALELA APLICADA AO PROCESSAMENTO DE IMAGENS E
COMPUTAÇÃO DE ALTO DESEMPENHO: UMA ABORDAGEM EXPERIMENTAL

Trabalho de conclusão de curso de graduação
apresentado ao Instituto de Computação da
Universidade Federal do Amazonas como parte
dos requisitos para obtenção do grau de
Bacharel em Ciência da Computação.

Orientador: Prof. Dr. João Marcos Bastos
Cavalcanti.

MANAUS

2025

RESUMO

Com o avanço das arquiteturas multicore e a crescente demanda por aplicações de alto desempenho, a programação paralela tornou-se uma abordagem essencial para acelerar a execução de algoritmos computacionais. Este trabalho de conclusão de curso apresenta uma investigação experimental sobre a aplicação de técnicas de programação paralela em diferentes contextos computacionais, com ênfase no processamento de imagens e técnicas computacionais de alto desempenho. A pesquisa é fundamentada em dois projetos de iniciação científica consecutivos (2023-2024 e 2024-2025), nos quais foram desenvolvidas e analisadas versões tradicionais e paralelas de algoritmos de processamento de imagens como transformações radiométricas, filtragens espaciais, detecção de bordas, além de aplicações mais gerais como multiplicação de matrizes, ordenação, busca e métodos numéricos. Foram utilizados diversos ambientes e linguagens de programação, incluindo Python, C, C com threads, OpenMP, Julia e Halide. Através de uma série de experimentos este trabalho compara o desempenho das abordagens sequenciais e paralelas em diferentes configurações de hardware, variando a quantidade de núcleos disponíveis na máquina, demonstrando os ganhos de eficiência, escalabilidade e viabilidade da paralelização. Os resultados obtidos evidenciam que a programação paralela, além de reduzir significativamente o tempo de execução em tarefas intensivas, é uma ferramenta estratégica para o desenvolvimento de soluções computacionais eficientes em cenários que exigem alto desempenho.

Palavras-chave: Programação Paralela, Processamento de Imagens, Alto Desempenho, Halide, OpenMP, Julia, Threads.

ABSTRACT

Recent advances in multicore architectures and the increasing demand for high-performance applications have made parallel programming an essential approach to accelerate the execution of computational algorithms. This work presents an experimental investigation into the application of parallel programming techniques in various computational contexts, with a focus on image processing and high-performance computing methods. This work is based on two consecutive undergraduate research projects (PIBIC 2023–2024 and 2024–2025), in which traditional and parallel versions of image processing algorithms were developed and analyzed, including radiometric transformations, spatial filtering and edge detection. In addition, other applications were studied such as matrix multiplication, sorting algorithms, searches, and numerical methods. Several programming environments and languages were used, including Python, C, C with threads, OpenMP, Julia and Halide. Through a series of experiments, this study compares the performance of sequential and parallel approaches across different hardware configurations, with varying numbers of CPU cores, demonstrating the gains in efficiency, scalability and feasibility of parallelization. The results show that parallel programming not only significantly reduces execution time in intensive tasks but also serves as a strategic tool for developing efficient computational solutions in high-performance scenarios.

Keywords: Parallel Programming, Image Processing, High Performance, Halide, OpenMP, Julia, Threads.

LISTA DE FIGURAS

1. Concorrência e paralelismo
2. Taxonomia de Flynn
3. Paralelismo em pixels
4. Paradigma data-parallel
5. Transformada logaritmo
6. Filtro da média
7. Detecção de bordas sobel
8. Limiarização de Otsu
9. Parallel matrizes
10. QuickSort
11. MergeSort
12. Busca em largura
13. Busca com múltiplas chaves
14. Redução de soma
15. Monte carlo para aproximação de pi
16. Parallel limiarização de Otsu
17. Código sequencial Otsu
18. Código paralelo Otsu
19. Código sequencial expansão linear
20. Código paralelo expansão linear
21. Código sequencial filtro da média
22. Código paralelo filtro da média
23. Código sequencial operador Sobel
24. Código paralelo operador Sobel
25. Código sequencial multiplicação de matrizes
26. Código paralelo multiplicação de matrizes
27. Código sequencial QuickSort
28. Código paralelo QuickSort
29. Código sequencial MergeSort
30. Código paralelo MergeSort
31. Código sequencial Busca em largura
32. Código paralelo Busca em largura
33. Código sequencial Busca com múltiplas chaves
34. Código paralelo Busca com múltiplas chaves
35. Código sequencial Redução de soma

36. Código paralelo Redução de soma
37. Código sequencial Monte carlo
38. Código paralelo Monte carlo
39. Resultados Limiarização de Otsu
40. Resultados Transformadas Radiométricas 4 threads
41. Resultados Transformadas Radiométricas 24 threads
42. Resultados Filtragens Espaciais 4 threads
43. Resultados Filtragens Espaciais 24 threads
44. Resultados Detecção de Bordas 4 threads
45. Resultados Detecção de Bordas 24 threads
46. Resultados multiplicação de matrizes
47. Resultados QuickSort
48. Resultados MergeSort
49. Resultados Busca em largura
50. Resultados Busca com múltiplas chaves
51. Resultados Redução de soma
52. Resultados Resultados Monte carlo

LISTA DE TABELAS

1. Taxonomia de Flynn
2. Fator avaliado x configuração avaliada
3. Ferramentas, Linguagens e Frameworks
4. Resultados Limiarização de Otsu
5. Resultados Transformadas Radiométricas 4 threads
6. Resultados Transformadas Radiométricas 24 threads
7. Resultados Filtragens Espaciais 4 threads
8. Resultados Filtragens Espaciais 24 threads
9. Resultados Detecção de Bordas 4 threads
10. Resultados Detecção de Bordas 24 threads
11. Configurações de Hardware
12. Resultados multiplicação de matrizes
13. Resultados algoritmos de ordenação
14. Resultados Busca em largura
15. Resultados Busca com múltiplas chaves
16. Resultados Redução de soma
17. Resultados Monte carlo

SUMÁRIO

1. INTRODUÇÃO	9
2. FUNDAMENTAÇÃO TEÓRICA	11
2.1. Conceitos Básicos de Programação Paralela	11
2.2. Fundamentos de Processamento de Imagens	14
2.3. Problemas Computacionais de alto desempenho	17
2.3.1. Multiplicação de matrizes	17
2.3.2. Ordenação	18
2.3.3. Processamento em grafos e buscas	19
2.3.4. Redução paralela (soma)	21
2.3.5. Simulação de Monte Carlo para aproximação de π	22
3. METODOLOGIA	22
3.1. Planejamento Experimental	22
3.2. Ferramentas, Linguagens e Frameworks	24
3.3. Conjunto de algoritmos escolhidos	25
3.3.1. Processamento de Imagens	25
3.3.1.1. Limiar de Otsu (segmentação por limiarização)	25
3.3.1.2. Transformações radiométricas	27
3.3.1.3. Filtragem espacial	28
3.3.1.4. Detecção de bordas	29
3.3.2. Problemas Computacionais de alto desempenho	30
3.3.2.1. Multiplicação de matrizes	30
3.3.2.2. Algoritmos de ordenação	31
3.3.2.3. Buscas em grafos (BFS), busca binária com múltiplas chaves	32
3.3.2.4. Redução paralela (soma)	34
3.3.2.5. Monte Carlo para aproximação de π	35
4. EXPERIMENTOS E RESULTADOS	35
4.1. Comparações de Desempenho em Processamento de Imagens	35
4.2. Comparações dos problemas computacionais de alto desempenho	44
4.3. Discussão dos Resultados	54
5. CONCLUSÃO	56
6. REFERÊNCIAS	58
7. APÊNDICES	60

1. INTRODUÇÃO

O avanço contínuo da tecnologia digital, especialmente no campo das arquiteturas multicore, tem impulsionado a necessidade por soluções computacionais capazes de lidar com volumes massivos de dados e algoritmos de alta complexidade. Nesse contexto, a programação paralela surge como uma estratégia fundamental para aproveitar o potencial de processamento distribuído entre múltiplos núcleos (GRAMA et al., 2003). Diversas áreas da computação, como aprendizado de máquina, ciência de dados, processamento de imagens, simulações científicas e desenvolvimento de jogos, têm recorrido a abordagens paralelas para alcançar maior desempenho e eficiência. Essa tendência acompanha a crescente demanda por aplicações rápidas, escaláveis e que aproveitem ao máximo os recursos disponíveis no hardware moderno.

Embora o processamento de imagens continue sendo um dos campos mais favorecidos pelo paralelismo, não é o único que se beneficia dessa abordagem. Operações como multiplicação de matrizes, algoritmos de ordenação (como Quick Sort e Merge Sort), buscas em estruturas de dados (como BFS e busca binária), métodos estatísticos (como Monte Carlo) e técnicas de redução paralela são igualmente favorecidas quando implementadas em ambientes paralelos. Essas técnicas envolvem tarefas repetitivas e independentes que se encaixam perfeitamente no modelo de divisão de trabalho da computação paralela, resultando em ganhos expressivos de desempenho. Portanto, explorar um conjunto diversificado de algoritmos oferece uma visão mais completa do impacto e aplicabilidade da programação paralela.

Resultados preliminares de dois projetos PIBIC, que serviram como base para este estudo, revelam ganhos que variam de 131 vezes até 1665 vezes entre as linguagens Halide e Python em diferentes técnicas de processamento de imagens, e de até 208 vezes mais rápidos que implementações tradicionais na linguagem C em filtros complexos. Essa variação indica que paralelizar nem sempre traz o mesmo retorno: depende do algoritmo, do número de núcleos e do overhead de implementação. Uma análise sistemática desses fatores, comparando Halide, OpenMP, threads em C e Julia, é portanto necessária para orientar escolhas de desenvolvedores e pesquisadores levando em consideração o problema explorado.

Além do mérito científico, o trabalho tenta proporcionar motivação econômica e social sobre questões de aplicabilidade em contextos reais da programação paralela. A Petrobras,

por exemplo, investiu R\$500 mi em cinco novos supercomputadores para manter a liderança em processamento sísmico, enquanto a IDC projeta que a falta de especialistas em TI custará US \$5,5 tri às organizações até 2026. Ao gerar código aberto (compartilhados no GitHUB), dados reproduutíveis e diretrizes práticas, este trabalho busca ajudar a fomentar informações que podem ser úteis para mão de obra qualificada, aplicabilidade da programação paralela em diversos contextos e utilização eficiente de hardware de acordo com o objetivo a ser alcançado.

Apesar da popularização de arquiteturas **multicore** em desktops, servidores e até dispositivos móveis, ainda falta uma **caracterização sistemática** que oriente o desenvolvedor sobre **quando e a que custo** adotar programação paralela em tarefas intensivas. Resultados preliminares experimentais mostram ganhos que variam de **2 vezes a mais de 1000 vezes** no tempo de execução, dependendo da técnica, linguagem e hardware usados. Mas também revelam overheads de implementação, curva de aprendizado e sensibilidade ao número de núcleos disponíveis.

Na área de **processamento de imagens**, a literatura já sugere forte afinidade com paralelismo por tratar essencialmente de matrizes de pixels independentes. Entretanto, quando se estende a análise para **outras operações de alto desempenho** — como multiplicação de matrizes, ordenação, buscas, redução paralela e métodos Monte Carlo — não há consenso sobre a magnitude dos ganhos nem sobre qual combinação *algoritmo × framework* entrega o melhor compromisso entre velocidade, portabilidade e esforço de desenvolvimento.

Diante desse contexto, formulamos o problema de pesquisa deste TCC:

- *Como quantificar e explicar o ganho de desempenho obtido pela programação paralela em um conjunto representativo de algoritmos de processamento de imagens e de computação de alto desempenho, avaliando o trade-off entre complexidade de implementação e escalabilidade em diferentes configurações de hardware?*

Pergunta(s) Central(is):

1. **Em que medida** as versões paralelas superam as versões sequenciais em **tempo de execução, uso de CPU e escalabilidade**, quando variamos:

- tamanho da entrada (tamanho da imagem, ordem da matriz, profundidade da busca, etc);
- número de núcleos de CPU disponíveis;

- categoria algorítmica (ponto-a-ponto, convolucionais, recursiva, numérica, etc).
2. **Quais fatores** (tipo de algoritmo, padrão de acesso à memória, característica do hardware) **mais influenciam** o speedup real, e **quais diretrizes práticas** podem ser extraídas para orientar futuras escolhas de ferramentas e arquitetura?

O objetivo deste trabalho é conduzir uma investigação experimental abrangente que compare versões sequenciais (Python e C) e paralelas (Halide, C + threads, OpenMP e Julia) de um conjunto de algoritmos de processamento de imagens e de computação de alto desempenho, quantificando ganhos de performance (tempo de execução, speedup e escalabilidade) e analisando o trade-off entre esforço de implementação e benefício obtido em diferentes arquiteturas multicore.

Para o desenvolvimento do trabalho seguimos as seguintes etapas:

- **Revisão bibliográfica** sobre modelos de programação paralela e métricas de avaliação de desempenho, mapeando técnicas de processamento de imagens e aplicações numéricas.
- **Selecionar e implementar** as versões sequenciais e paralelas dos algoritmos-alvo:
 - **Processamento de Imagens** – transformações radiométricas (log, dente de serra, expansão de contraste), filtragens espaciais (média, mediana, K-vizinhos); detecção de bordas (Sobel, Roberts) e limiarização de Otsu;
 - **Problemas adicionais** – multiplicação de matrizes, Quick Sort, Merge Sort, BFS, busca binária multichave, redução paralela (soma) e Monte Carlo para π ;
 - **Analizar criticamente** os resultados, identificando fatores que mais influenciam o speedup e o ponto em que o custo de paralelizar deixa de compensar.
 - **Elaborar diretrizes práticas** para desenvolvedores e pesquisadores, indicando em quais cenários cada ferramenta ou abordagem paralela apresenta a melhor relação custo-benefício.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. Conceitos Básicos de Programação Paralela

Paralelismo × Concorrência – Concorrência é a capacidade de progredir em várias tarefas alternando a atenção do processador; paralelismo é a execução simultânea dessas tarefas em núcleos diferentes. A figura 1 (concorrência e paralelismo) ilustra isso: na concorrência a Task 1 e 2 disputam um único núcleo, enquanto na de baixo elas correm lado a lado em dois núcleos diferentes (paralelismo). Essa distinção é fundamental porque todo

programa paralelo é, de certa forma, concorrente, mas nem todo programa concorrente alcança execução realmente simultânea.

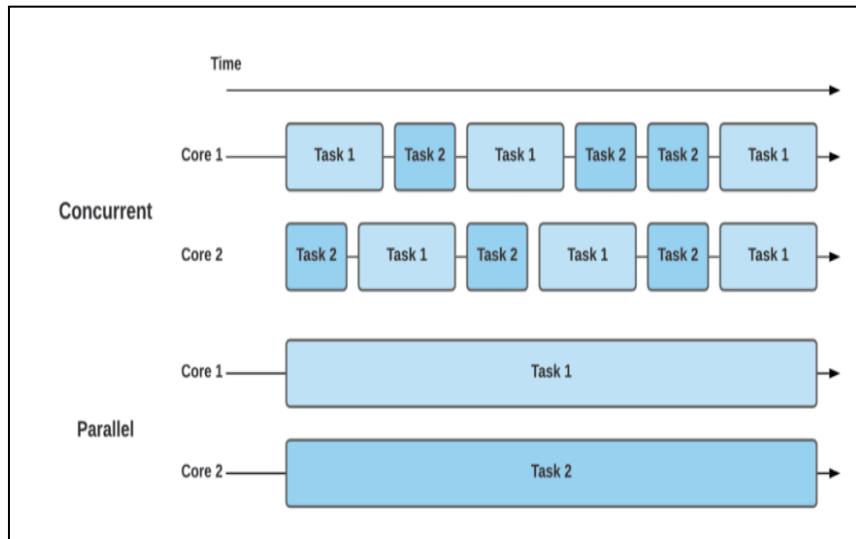


Figura 1: Concorrência e paralelismo

Para quantificar o ganho usamos o speedup, definido por:

$$S(p) = \frac{T_{seq}}{T_{par}},$$

onde T_{seq} é o tempo sequencial e T_{par} é o tempo com p processadores. A eficiência $E(p) = S(p)/p$ valores próximos de 1 representam bom aproveitamento dos cores. Esses conceitos são fundamentais na avaliação de desempenho em sistemas paralelos e estão bem estabelecidos na literatura de computação de alto desempenho (HENNESSY; PATTERSON, 2019)

Taxonomia de Flynn — SISD, SIMD, MISD, MIMD — Michael Flynn classificou os computadores segundo fluxo de instruções e fluxo de dados:

Classe	Instruções	Dados	Exemplo
SISD	1	1	Microcontrolador
SIMD	1	N	GPU, AVX
MISD	N	1	Pipelines tolerantes a falha
MIMD	N	N	CPUs multicore, clusters

Tabela 1: Taxonomia de Flynn

A figura 2 (Taxonomia de Flynn) exemplifica o processo descrito na tabela acima, em quatro quadrantes, como instruções (barras vermelhas) e dados (barras roxas) se

relacionam com os processadores (blocos verdes): **SISD** mostra um único processador executando uma única instrução sobre um único fluxo de dados; **SIMD** réplica vários processadores que aplicam a mesma instrução em diferentes partes do mesmo conjunto de dados, típico de GPUs; **MISD** encadeia processadores que executam instruções distintas sobre o mesmo dado para verificação de falhas; e **MIMD** apresenta diversos processadores independentes, cada um com suas próprias instruções e dados, como em clusters ou CPUs multicore.

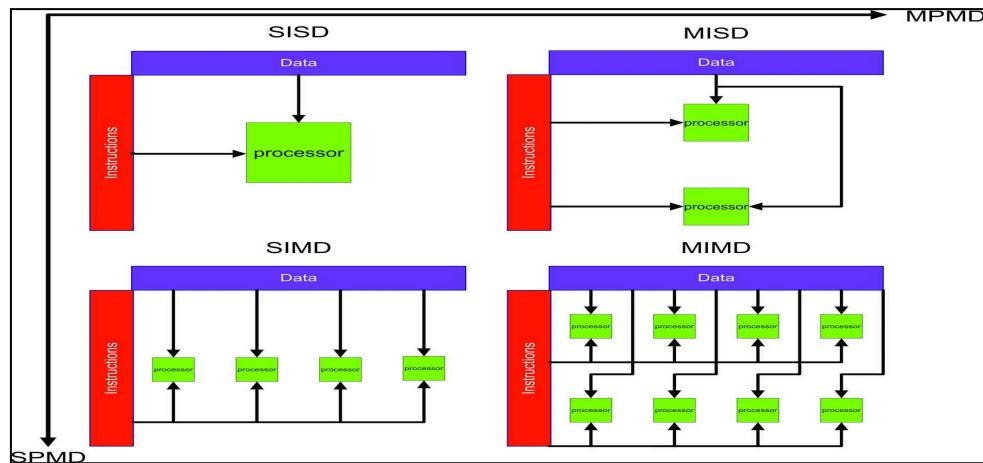


Figura 2: Taxonomia de Flynn

Do sequencial ao paralelo — Imagine aplicar um ajuste de brilho a uma foto de 8 MP. No código sequencial, um laço percorre **pixel a pixel**: cada entrada é lida, processada e gravada antes de passar à próxima, conforme ilustrado na figura abaixo:

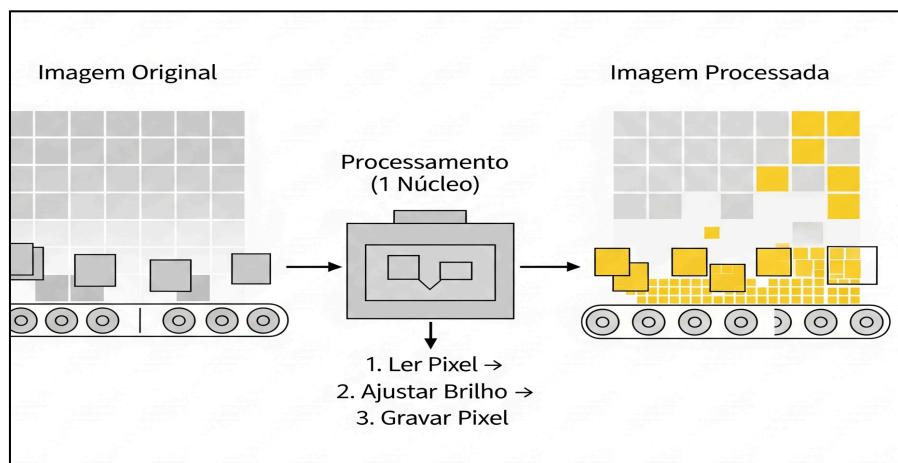


Figura 3: Paralelismo em pixels

No paradigma **data-parallel**, dividimos a matriz de pixels em “fatias” (tiles) e distribuímos cada tile a um núcleo ou thread, que repete a mesma operação em paralelo. Ao

final, os resultados são reunidos (redução) formando a imagem de saída. Esse padrão de divisão-e-conquista generaliza-se para matrizes (multiplicação em blocos), listas (quicksort paralelo) e grafos (BFS em níveis). O Processo descrito acima é exemplificado pela figura abaixo:

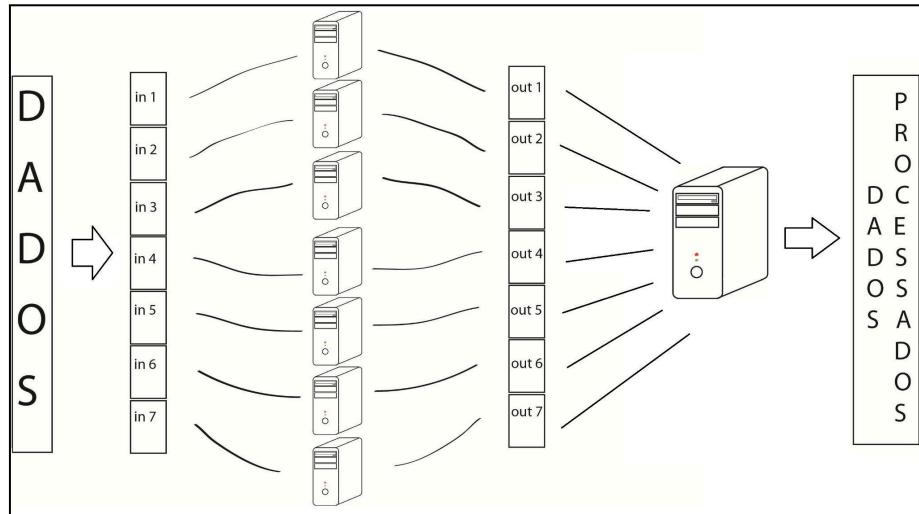


Figura 4: Paradigma data-parallel

Os conceitos discutidos nesta seção fornecem uma base teórica para entender os fundamentos e conceitos básicos referente à programação concorrente e paralela, fornecendo utensílios para maior compreensão das próximas etapas que este trabalho tem a apresentar, como escolhas de técnicas e métricas de análises.

2.2. Fundamentos de Processamento de Imagens

O ponto de partida de qualquer técnica está na representação matricial de uma imagem: um vetor bidimensional $I = (x, y)$ em que cada elemento armazena a intensidade luminosa daquele pixel. Em processamento de imagens, operações típicas envolvem as técnicas de: Transformadas Radiométricas, Filtragem Espacial e Detecção de Bordas.

Transformações radiométricas manipulam os valores de intensidade para ajustar brilho ou contraste sem alterar a posição dos pixels, sendo amplamente descritas na literatura clássica de processamento de imagens (GONZALEZ; WOODS, 2018); a forma mais simples é a linear $I(x, y) = a I(x, y) + b$ onde a controla ganho e b desloca o nível médio — útil para corrigir imagens sub- ou super expostas e para normalizar faixas de cinza antes de operações mais complexas. A imagem abaixo exemplifica a operação logarítmica (solução radiométrica) sendo aplicada em uma imagem.

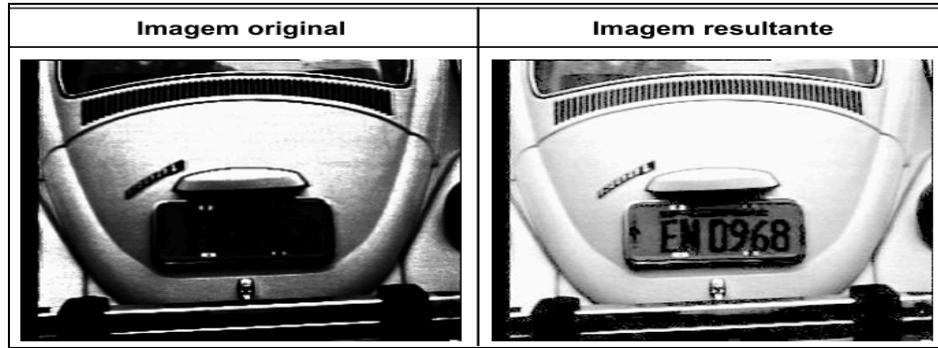


Figura 5: Transformada logaritmo

Já os **filtros espaciais** analisam a vizinhança de cada pixel aplicando a operação de convolução, técnica amplamente abordada por Gonzalez e Woods (2018):

$$g(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k \omega(i, j)f(x - i, y - j),$$

na qual $\omega(i, j)$ é o kernel (máscara) e o k define o raio da janela. Valores positivos e negativos em ω permitem suavizar ruídos (média, gauss) ou realçar detalhes (Laplaciano). Como cada posição (x, y) depende apenas de seus vizinhos imediatos, a convolução é intrinsecamente data-parallel: basta repartir blocos da matriz entre núcleos sem risco de concorrência, desde que as bordas dos blocos recebam círculo de luz adequado. Abaixo segue uma imagem na qual foi aplicado o filtro da média para tirar ruídos do tipo sal e pimenta.



Figura 6: Filtro da média

A **detecção de bordas** procura transições bruscas de intensidade examinando o gradiente da imagem. Métodos clássicos como os operadores de Sobel e Roberts são detalhados em Gonzalez e Woods (2018). O operador de Sobel calcula derivadas discretas

nas direções x e y com dois kernels G_x e G_y a magnitude resultante é $\|\nabla I\| = \sqrt{G_x^2 + G_y^2}$.

Esse passo destaca contornos e serve de base para segmentação ou reconhecimento de objetos. Na prática, o método equilibra simplicidade computacional e boa resposta a ruído, tornando-se escolha clássica em pré-processamento de sistemas de visão, robótica móvel e análise médica. A imagem abaixo exemplifica a operação de detecção de bordas pelo operador Sobel, comparando a imagem original com a resultante que realça os contornos.

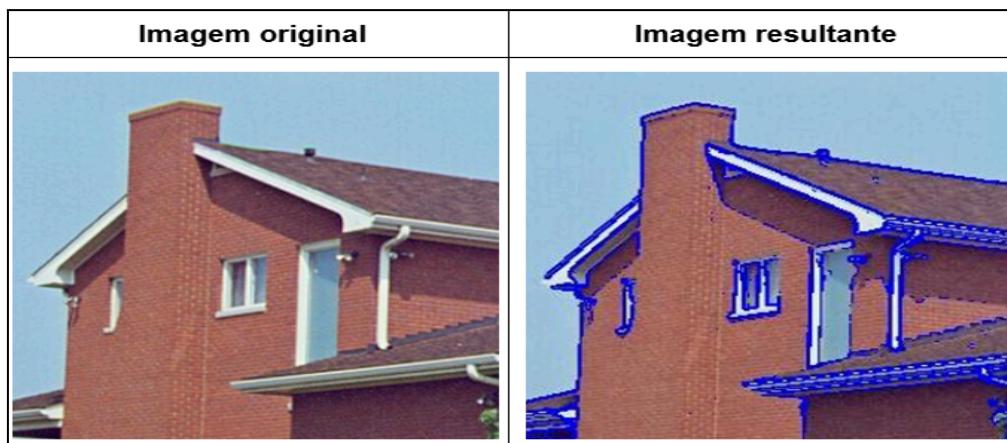


Figura 7: Detecção de Bordas Sobel

O método de Otsu O método de Otsu, como apresentado por Gonzalez e Woods (2018), procura um limiar que maximize a separação estatística entre primeiro plano e fundo em histogramas de níveis de cinza $p(i) = (i = 0 \dots L - 1)$, um limiar t^* que maximiza a separação estatística entre primeiro plano e fundo. Para cada candidato t calcula-se: (i) as

probabilidades acumuladas $\omega_0(t) = \sum_{i=0}^t p(i)$ e $\omega_1(t) = 1 - \omega_0(t)$; (ii) as médias $\mu_0(t)$ e $\mu_1(t)$; e (iii) a variância entre as classes $\sigma_b^2(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$.

O limiar ótimo é $t^* = \arg \max_t \sigma_b^2(t)$, equivalente a minimizar a variância intra-classe $\sigma_w^2(t)$. Esse critério é um análogo discreto da análise discriminante de Fisher e funciona bem para histogramas bimodais, gerando automaticamente uma imagem binária sem parâmetros empíricos.

A imagem abaixo representa o processo descrito acima, à esquerda, a cena original de Saturno e, à direita, o resultado da segmentação binária obtida pelo método de Otsu, realçando o limiar automático entre primeiro plano e fundo.

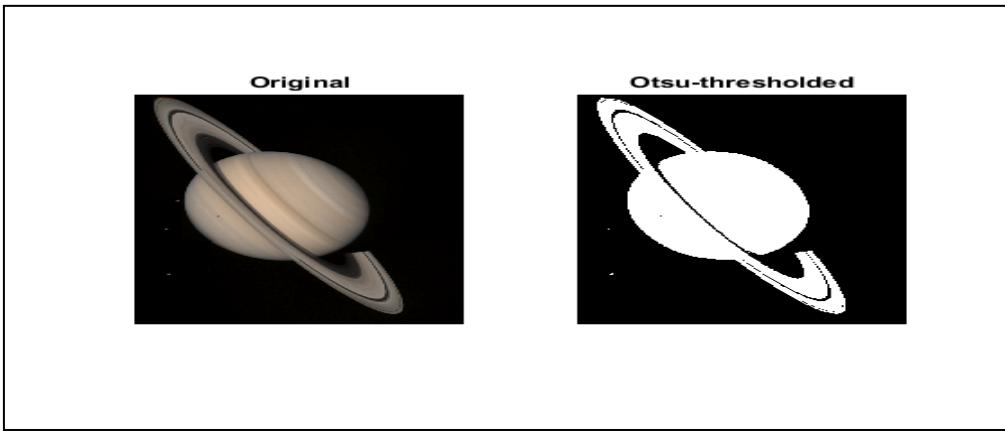


Figura 8: Limiarização de Otsu

Do ponto de vista computacional, o algoritmo exige apenas duas passagens lineares: uma varredura na imagem para montar o histograma (paralelizável por partição de blocos) e outra sobre o histograma — de tamanho fixo $L \leq 256$ para acumular ω , μ e encontrar t^* .

2.3. Problemas Computacionais de alto desempenho

2.3.1. Multiplicação de matrizes

A multiplicação de matrizes é um problema fundamental na álgebra linear computacional e serve como benchmark para avaliação de desempenho (GILBERT; MOLER; SCHREIBER, 1992).

A forma clássica se compõe em multiplicar $A = (n \times m)$ e $B = (m \times p)$ produz $C = AB$ cujos elementos são:

$$c_{ij} = \sum_{k=1}^m a_{ik} a_{kj}, \quad 1 \leq i \leq n, \quad 1 \leq j \leq p,$$

exigindo $n m p$ multiplicações-adições- n^3 no caso cúbico. Esse algoritmo serve de referência para métodos mais rápidos (Strassen reduz para $O(n^{2.807})$), mas continua predominante em faixas de tamanho moderado por manter estabilidade numérica e aproveitar vetorização automática dos compiladores.

O gargalo moderno é o trânsito de dados entre memória e cache. A estratégia blocked (tiled) GEMM, ilustrada na figura 8, partitiona as matrizes em sub-blocos $B \times B$ que cabem no cache; cada bloco A_{pq} combina-se com B_{qr} para gerar C_{pr} . Isso reduz cache misses de $O(n^3)$ para $O(\frac{n^3}{B})$ e como, cada C_{pr} é exclusivo de um bloco, elimina escrita

concorrente. Assim, laços externos podem ser distribuídos por threads OpenMP, kernels CUDA ou agendados, alcançando escalabilidade quase linear em CPUs multicore ou GPUs.

$$\begin{aligned}
 A &= \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \end{array} \right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \\
 A &= \left[\begin{array}{cccc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{array} \right] = \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \mathbf{r}_3 \end{bmatrix} \\
 A &= \left[\begin{array}{c|c|c|c} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{array} \right] = [\mathbf{c}_1 \quad \mathbf{c}_2 \quad \mathbf{c}_3 \quad \mathbf{c}_4]
 \end{aligned}$$

Figura 9: Parallel matrices

Uma matriz pode ser subdividida em blocos ou particionada em matrizes menores inserindo cortes horizontais e verticais entre linhas e colunas conforme ilustrado na Figura 9 (Parallel matrices).

2.3.2. Ordenação

Quicksort aplica Dividir-para-Conquistar: escolhe um pivô, partitiona o vetor em elementos menores e maiores, e então recursivamente ordena cada parte. A recorrência $T(n) = T(k) + T(n - k - 1) + O(n)$ leva ao tempo médio $O(n \log n)$ e pior caso $O(n^2)$ se o pivô for mal escolhido; a pilha de chamadas consome $O(\log n)$ memória na média. Depois que a fase de partitionamento termina, os dois sub-vetores são independentes — característica que permite paralelizar o algoritmo delegando cada sub-problema a uma thread ou task. A imagem abaixo descreve o passo a passo do algoritmo em sua versão sequencial.

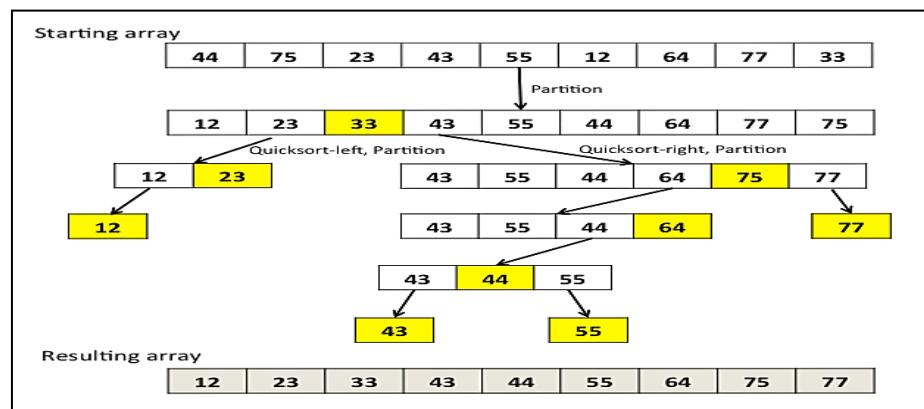


Figura 10: QuickSort

Merge Sort divide o vetor exatamente ao meio, ordena as metades e as funde numa sequência ordenada. A recorrência $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ resolve em $T(n) = \Theta(n \log n)$ em todos os casos; o algoritmo é estável e fácil de provar correto, mas requer $O(n)$ memória extra para armazenar o vetor temporário de mesclagem. Como cada metade é tratada de forma autônoma, é comum paralelizar a etapa de ordenação recursiva (uma thread por metade) e, em implementações mais avançadas, também o próprio passo de merge via algoritmos de mesclagem paralela, obtendo bom speedup em sistemas multicore.

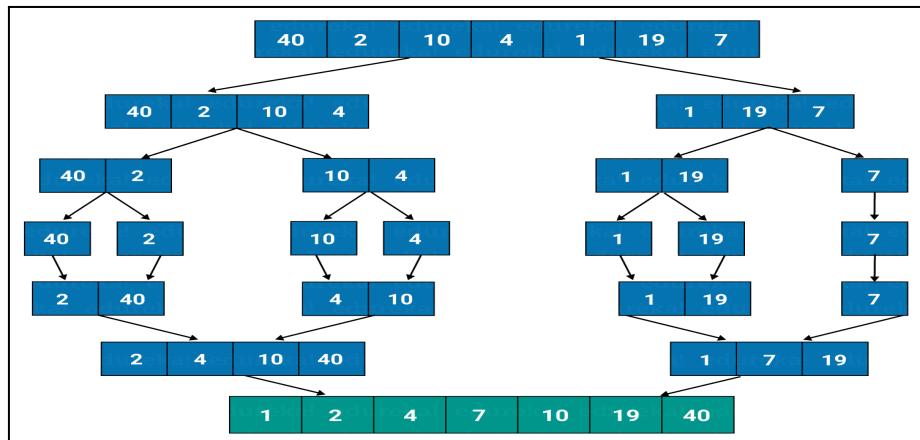


Figura 1: MergeSort

A imagem acima exemplifica o Merge Sort em suas versão sequencial, mostrando o vetor inicial sendo dividido recursivamente ao meio até sub vetores unitários que são então fundidos em blocos ordenados.

2.3.3.Processamento em grafos e buscas

Busca em Largura (BFS) percorre o grafo nível-a-nível: parte do vértice-fonte, coloca-o numa fila FIFO, depois expande todos os vizinhos, marcando cada vértice visitado para evitar ciclos. A cada remoção da fila processamos no máximo todas as arestas incidentes, somando $O(V + E)$ visitas - V vértices, E arestas - portanto o algoritmo é linear no tamanho do vértice. Como cada camada (fronteira) é independente da seguinte, implementações paralelas dividem o vetor de vértices da camada corrente entre threads.

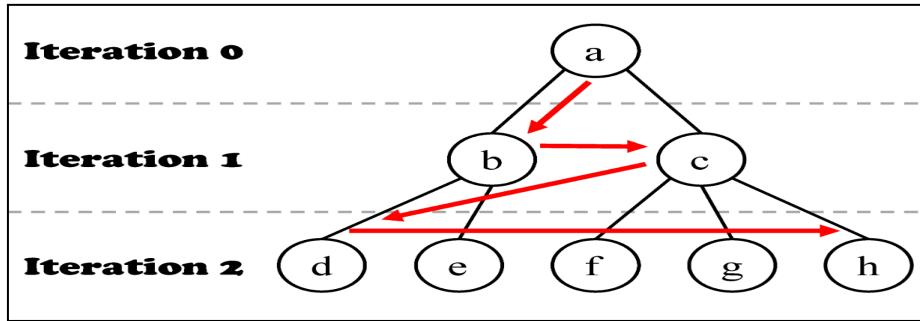


Figura 12: Busca em largura

A **Figura 12** (Busca em largura) ilustra a execução do BFS em três iterações de fronteiras — processando inicialmente o vértice fonte, depois sua camada de vizinhos imediatos e, por fim, a expansão dessa camada subsequente.

Busca Binária, apesar de ser intrinsecamente sequencial, pode ser paralelizada em cenários com múltiplas consultas (KNUTH, 1998). A técnica opera sobre um vetor ordenado, comparando o elemento-meio com a chave: se menor, descarta a metade esquerda; se maior, descarta a direita. A recorrência $T(n) = T(\frac{n}{2}) + O(1)$ resolve-se em $T(n) = \Theta(\log n)$ comparações. Embora a própria busca seja intrinsecamente sequencial, pode-se paralelizar cenários com múltiplas chaves: cada thread executa uma busca independente sobre o mesmo vetor, evitando contenção porque apenas leituras ocorrem.

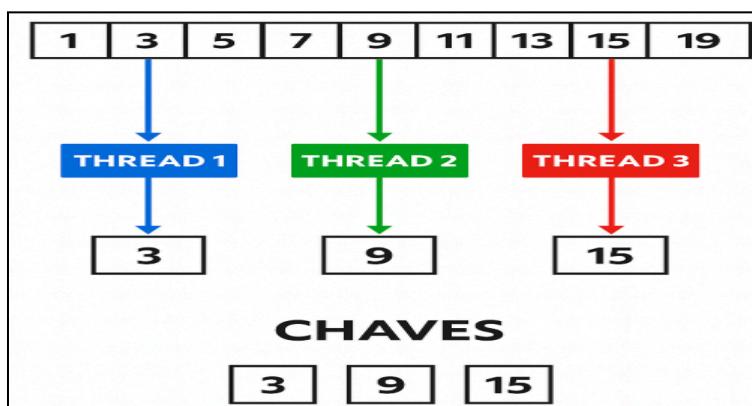


Figura 13: Busca com múltiplas chaves

A **figura 13** (Busca binária com múltiplas chaves) ilustra buscas binárias paralelas de múltiplas chaves em um vetor ordenado, com cada thread realizando sua própria pesquisa de forma independente.

Essas duas técnicas ilustram como a natureza dos dados determina o potencial de paralelismo: BFS oferece paralelismo entre vértices de uma mesma camada, enquanto a busca binária só ganha quando há muitas consultas em paralelo.

2.3.4. Redução paralela (soma)

A redução é a operação de condensar um vetor v e n elementos em um único resultado aplicando uma função associativa — no nosso caso a soma:

$$S = \sum_{i=0}^{n-1} v_i$$

No algoritmo sequencial o tempo é $T_{seq} = o(n)$; já a propriedade de associatividade $((a + b) + c = a + (b + c))$ permite reordenar os termos sem mudar o resultado, condição necessária para executar partes independentes em paralelo.

Para explorar todos os núcleos, bibliotecas de GPU e CPUs seguem o esquema exibido na **figura 14** (Redução de soma): um arbusto binário em que cada nível soma pares de elementos. Cada passo reduz o número de valores pela metade, obtendo tempo paralelo ideal $T_{par}(p) \approx \frac{n}{p} + \log_2 p$, onde o primeiro termo corresponde ao trabalho dividido entre p threads e o segundo ao número de níveis da árvore.

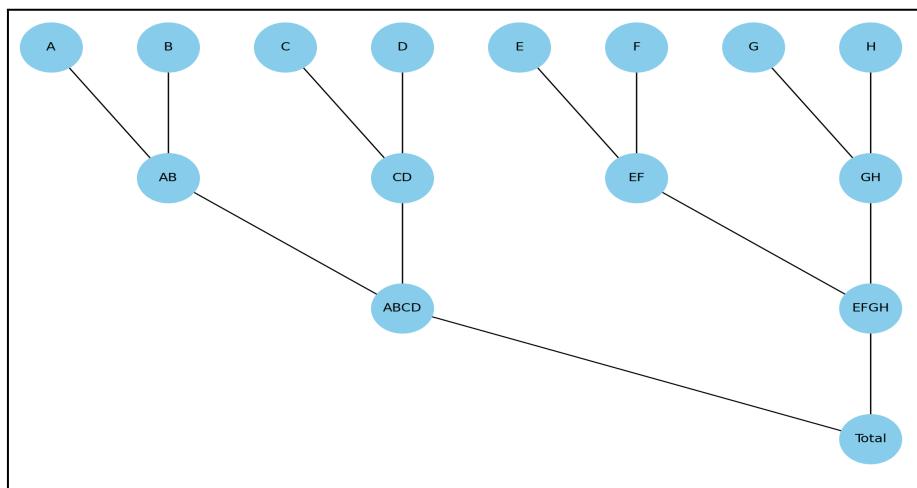


Figura 14: Redução de soma

Assim, a redução paralela ilustra como um simples princípio matemático — associatividade — se traduz em ganho quase linear de desempenho quando mapeado corretamente ao hardware.

2.3.5. Simulação de Monte Carlo para aproximação de π

A técnica de Monte Carlo é uma das primeiras aplicações de computação de alto desempenho e simulações estatísticas (METROPOLIS et al., 1953). O Problema considera um quadrado de lado 2 e, inscrito nele, um círculo de raio 1. O algoritmo de Monte Carlo lança N pontos pseudo-aleatórios (x, y) com $x, y \in [-1, 1]$ e conta quantos caem dentro do círculo, isto é, satisfazem $x^2 + y^2 \leq 1$. Se N_{in} a contagem é, então $\pi \approx 4 \frac{N_{in}}{N}$, pois a razão entre as áreas do círculo (πr^2) e do quadrado ($4r^2$) é $\frac{\pi}{4}$ para $r = 1$. O erro estatístico decai como $\sigma[\pi] \sim \frac{1}{\sqrt{N}}$, característica típica de amostragem aleatória, tornando a técnica simples porém convergente. A imagem abaixo representa o processo descrito:

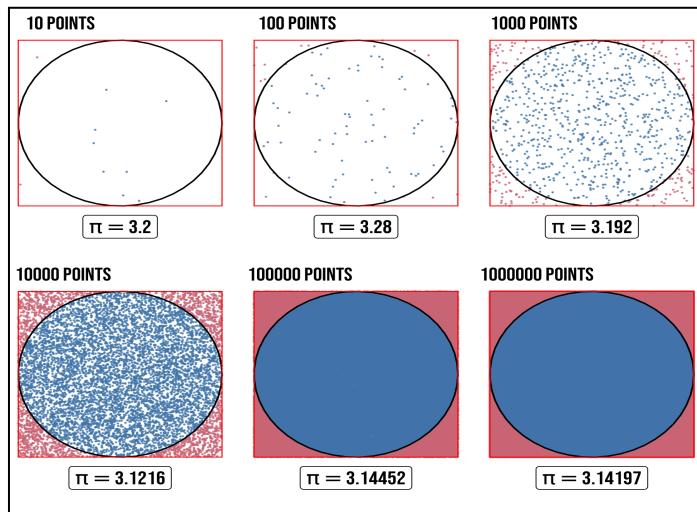


Figura 15: Monte carlo para aproximação de pi

Cada ponto é gerado e testado independentemente; logo, basta repartir os lançamentos entre múltiplos núcleos ou GPUs, acumular os contadores locais e fazer uma redução final $N_{in}^{total} = \sum p N_{in}^{(p)}$.

Assim, a aproximação de π ilustra claramente como problemas “embarrassingly parallel” maximizam o retorno da programação paralela.

3. METODOLOGIA

3.1. Planejamento Experimental

Para quantificar o ganho de desempenho das versões paralelas, foi adotado práticas de benchmarking apresentadas em guias de HPC, adotando tempo médio $u = \left(\frac{1}{n}\right) \sum_{i=1}^n t_i$

, onde t_i é o tempo da i -ésima medição; speed-up médio $S = \frac{u_{seq}}{u_{par}}$, sendo u_{seq} o tempo médio da versão sequencial e u_{par} o da versão paralela. Esses indicadores - u para comparar tempos absolutos e S para quantificar o ganho global - são os valores reportados nas tabelas e gráficos das seções de Resultados. A tabela abaixo apresenta o fator avaliado e a configuração adotada no experimento.

Fator avaliado	Configuração Avaliada
Algoritmos	Processamento de imagens → Radiométricas, Filtragens, Detecção de Bordas e Limiar de Otsu; Problemas gerais → multiplicação de matrizes, Ordenação, Buscas, Redução e Monte Carlo.
Implementações	Sequencial → Python, C; Paralelo → C+Threads, OpenMP, Halide, Julia.
Núcleos ativos	1 - 4 na CPU i5-7200U; 1 - 6 na CPU i5-9500; 1 - 16 na CPU i5-12500H; 1 - 24 na CPU i7-13700.

Tabela 2: Fator avaliado x configuração avaliada

Os quatro fatores da tabela se concentram no que realmente influencia a medição: **algoritmo**, para verificar padrões de acesso à memória diversos; **implementação**, para comparar os graus de paralelismo e níveis de abstração; **núcleos ativos**, verificando o efeito da arquitetura na escalabilidade.

Para as operações de processamento de imagens selecionamos um conjunto de 30 imagens de tamanhos variados para a realização dos experimentos. Essas imagens foram organizadas em seis grupos, categorizados de acordo com a quantidade de pixels. O primeiro grupo possui entre 1.800.000 a 1.900.000 pixels, o segundo grupo entre 2.000.000 a 4.000.000 pixels, o terceiro grupo entre 4.000.000 a 8.000.000 pixels, o quarto grupo entre 9.000.000 a 12.200.000 pixels, o quinto grupo entre 12.200.000 a 19.000.000 pixels e o sexto grupo entre 22.000.000 a 33.000.000 pixels.

Para as operações de problemas gerais, como multiplicação de matrizes, algoritmos de ordenação, busca binária, busca em grafos e a técnica de Monte Carlo, foram utilizados vetores e matrizes com tamanhos variados. Esses tamanhos foram definidos de forma crescente, visando observar como o desempenho das abordagens se comporta frente ao aumento da carga computacional.

No caso da multiplicação de matrizes, os tamanhos variaram de matrizes 1.000 x 1.000 até 10.000 x 10.000. Para os algoritmos de ordenação, busca e redução, foram utilizados vetores que variaram de 10 milhões até 500 milhões de elementos, dependendo do problema. Essa variação permitiu avaliar não apenas o impacto do tamanho dos dados, mas também a escalabilidade das técnicas paralelas empregadas nas diferentes arquiteturas de hardware testadas.

Com as etapas experimentais amarradas, passamos para as **Ferramentas, Linguagens e Frameworks**, onde é especificado as particularidades de cada recurso utilizado, verificando que bibliotecas e padrão de codificação poderiam ser explorados para escrita de algoritmos paralelos e tradicionais.

3.2. Ferramentas, Linguagens e Frameworks

Abaixo segue a tabela contendo as principais ferramentas, linguagens e frameworks utilizados para desenvolvimento dos algoritmos:

Camadas/Ferramenta	Linguagem/Framework	Paralelismo e Otimização	Controle de Sincronismo
DLS para imagens	Halide (C++ embarcado)	Paralelismo automático via parallel() e vetorização com vectorize() . Otimização de cache e pipelines.	Sem locks manuais. Runtime garante sincronização apenas onde há dependências.
API de diretivas	OpenMP (C/C++/Fortran)	Paraleliza laços com #pragma omp parallel for . Suporte a SIMD.	Constructs como <i>critical</i> , <i>atomic</i> , <i>reduction</i> e <i>barrier</i> .
Biblioteca de baixo nível	POSIX pthreads (C)	Controle manual sobre criação de threads e divisão das tarefas.	Sincronismo explícito com <i>mutex</i> , <i>condition variables</i> e <i>spinlocks</i> .
linguagem GIT de alto nível compilada	Julia	Compila para LLVM. Usa @threads para laços e agendamento eficiente de tarefas.	Suporte à <i>Channels</i> , <i>Atomic</i> , <i>Locks</i> e detecção de data race.
Baseline sequencial (interpretada)	Python 3	Usado para orquestração e baseline. Processamento nativo sequencial.	Não é aplicável (execução sequencial).
Baseline sequencial (compilado)	C	Controle manual do uso de cache, memória e dados.	Não é aplicável (execução sequencial).

Tabela 3: Ferramentas, Linguagens e Frameworks

A tabela descreve como as linguagens utilizadas tratam os recursos que cada uma dispõe e como o sincronismo de threads é feito para não haver concorrência de tarefas, exceto para as linguagens que são exclusivamente para o processo sequencial.

3.3. Conjunto de algoritmos escolhidos

Os algoritmos selecionados neste trabalho foram escolhidos com base nos princípios da programação paralela, que consiste em dividir um problema em partes menores e executá-las simultaneamente em múltiplos núcleos ou processadores, como é o caso do processamento de imagens, que envolve manipulação de matrizes e permite paralelismo por divisão de dados.

Além disso, outros problemas como multiplicação de matrizes, algoritmos de ordenação (Quick Sort e Merge Sort), busca em largura (BFS) em grafos e técnicas embarrassingly parallel, como redução de soma e simulação de Monte Carlo, também possuem características que favorecem o paralelismo.

Todas essas técnicas discutidas na seção de fundamentação teórica foram implementadas utilizando diferentes abordagens, como OpenMP, Pthreads, Halide e Julia, que permitem explorar eficientemente os recursos paralelos disponíveis conforme descrito na tabela 3. Na próxima seção, são detalhadas as estratégias adotadas para cada problema.

3.3.1. Processamento de Imagens

3.3.1.1. Limiar de Otsu (segmentação por limiarização)

O processo envolve duas etapas principais: cálculo do histograma e aplicação do limiar. Nos códigos desenvolvidos, o paralelismo foi aplicado no cálculo do histograma e binarização, onde os pixels são distribuídos entre as threads, que constroem histogramas locais posteriormente combinados. A imagem abaixo exemplifica o processo adotado.

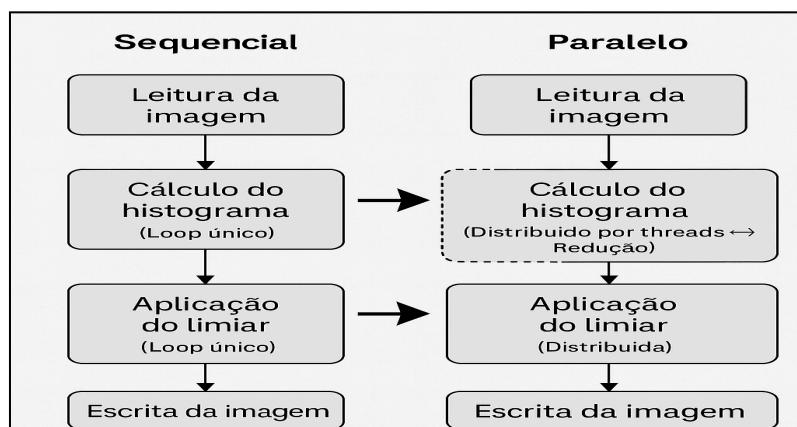


Figura 16: Parallel limiarização de Otsu

O trecho de código representado na figura abaixo é a implementação sequencial do problema:

```
/* 1) HISTOGRAMA — percorre todos os WxH pixels */
int hist[256] = {0};
for (int p = 0; p < W*H; p++) {
    hist[pixels[p]]++; // soma 1 no nível de cinza lido

/* 2) LÓGICA DE OTSU — 256 passos, custo desprezível */
double wB = 0, sumB = 0, varMax = 0; int T = 0;
for (int t = 0; t < 256; t++) { // testa cada limiar t
    wB += hist[t]; // peso da classe "fundo"
    sumB += t * hist[t]; // soma ponderada do fundo
    double wF = total - wB; // peso da "frente"
    if (wB == 0 || wF == 0) continue; // evita divisão por zero
    double mB = sumB / wB; // média do fundo
    double mF = (sumTotal - sumB) / wF; // média da frente
    double var = wB * wF * (mB - mF)*(mB - mF); // variância entre classes
    if (var > varMax) { varMax = var; T = t; } // guarda melhor t
}
```

Figura 17: código sequencial Otsu

O primeiro laço visita cada pixel em uma thread. O segundo laço visita só 256 níveis; para cada t calcula as estatísticas acumuladas e guarda o limiar de maior variância. Esse passo é constante em relação ao tamanho da imagem.

A sua versão paralela, em Halide, segue a seguinte nomenclatura:

```
/* HISTOGRAMA paralelizado por Halide */
Func hist("hist"); Var i;
hist(i) = cast<int32_t>(0); // zera 256 posições

RDom r(0, W, 0, H); // domínio = todos os pixels
hist(gray(r.x, r.y)) += 1; // incremento atômico

Buffer<int32_t> H = hist.realize({256}); // runtime divide por tiles

/* BINARIZAÇÃO paralela + SIMD */
Func bin("bin"); Var x, y;
bin(x,y) = select(gray(x,y) > T, 255, 0);
bin.vectorize(x, 8) // 8 pixels por instrução SIMD
| .parallel(y); // cada linha em uma thread
```

Figura 18: código paralelo Otsu

Para o histograma, Halide transforma a redução $+=$ sobre r em um kernel que atribui cada bloco de pixels a um núcleo e usa contadores privados, fundindo-os no fim. A binarização ocorre por meio de $parallel(y)$ distribui linhas entre núcleos e $vectorize(x, 8)$ gera instruções SIMD; como cada thread grava linhas distintas, não há colisão.

3.3.1.2. Transformações radiométricas

Neste trabalho usamos três mapeamentos clássicos - **logarítmico** (realçar tons escuros), **dente de serra** (enfatiza faixas alternadas) e **expansão linear de contraste** (estica o histograma entre dois limiares).

Adiante é destacado trechos de códigos da implementação da expansão linear em versão sequencial C e Halide (paralela). Os blocos destacam apenas onde o trabalho se torna paralelo ou permanece sequencial.

```

for (int i = 0; i < altura; i++) {
    for (int j = 0; j < largura; j++) {
        int pixel = imagem.at<uchar>(i, j);
        if (pixel <= za) {
            imagem_expandida.at<uchar>(i, j) = z1;
        } else if (pixel >= zb) {
            imagem_expandida.at<uchar>(i, j) = zn;
        } else {
            imagem_expandida.at<uchar>(i, j) = saturate_cast<uchar>(a * pixel + b);
        }
    }
}

```

Figura 19: código sequencial expansão linear

Na versão sequencial (Figura 19) todo o processamento ocorre em uma única thread que percorre os pixels linha-a-linha e coluna-a-coluna. Assim, cada pixel é processado um após o outro no mesmo núcleo.

Em Halide o pipeline é descrito de forma funcional; depois um schedule indica como o trabalho deve ser particionado:

```

Var x("x"), y("y"), c("c");
Func expansaoLinear("expansaoLinear");
Expr valor = input(x, y, c);
Expr transformada = select(valor <= za, cast<uint8_t>(z1),
                           valor >= zb, cast<uint8_t>(zn),
                           cast<uint8_t>(a * valor + b));
expansaoLinear(x, y, c) = transformada;
expansaoLinear.parallel(y).vectorize(x, 16);

```

Figura 20: código paralelo expansão linear

O `parallel(y)` distribui linhas independentes pelos núcleos, enquanto `vectorize(x, 16)` gera instruções SIMD que processam 16 pixels de uma vez. Como não há dependências

horizontais entre linhas, não é necessário sincronismo explícito; o runtime de Halide garante que cada thread escreva em regiões diferentes do buffer de saída, evitando *data races*.

3.3.1.3. Filtragem espacial

Foram implementadas as variações da **média** (suaviza ruídos), **mediana** (remove ruídos do tipo 'sal e pimenta') e **K-vizinhos mais próximos** (pondera os K pixels vizinhos). Como a operação é repetida uniformemente sobre a imagem, o problema é naturalmente paralelizável por linhas ou blocos independentes.

```

for (int i = 1; i < altura-1; i++)
    for (int j = 1; j < largura-1; j++) {
        Vec3f media(0,0,0);

        for (int x = -1; x <= 1; x++)
            for (int y = -1; y <= 1; y++) {
                Vec3b p = imagem.at<Vec3b>(i+x, j+y);
                media[0] += p[0]; media[1] += p[1]; media[2] += p[2];
            }

        media /= 9.0f;
        Vec3b pc = imagem.at<Vec3b>(i,j);
        if (fabs(media[0]-pc[0]) > T) imagemFiltrada.at<Vec3b>(i,j)[0]=(uchar)media[0];
        if (fabs(media[1]-pc[1]) > T) imagemFiltrada.at<Vec3b>(i,j)[1]=(uchar)media[1];
        if (fabs(media[2]-pc[2]) > T) imagemFiltrada.at<Vec3b>(i,j)[2]=(uchar)media[2];
    }
}

```

Figura 21: código sequencial filtro da média

A **figura 21** representa a versão sequencial do algoritmo do **filtro da média**, todos os quatro laços ficam na mesma thread, o algoritmo visita cada pixel uma após a outra, caracterizando execução totalmente sequencial.

Na versão em Halide o estágio de convolução é escrito de forma declarativa e depois agendado:

```

Var x("x"), y("y"), c("c");
Func mediaFiltro("mediaFiltro");

// entrada com repetição de borda
Func inputBound = BoundaryConditions::repeat_edge(input);

// janela 3x3 resumida por RDom
RDom janela(-1, 3, -1, 3);

// soma e média
Expr soma = sum(cast<float>(inputBound(x+janela.x, y+janela.y, c)));
Expr media = soma / 9.0f;

mediaFiltro(x,y,c) = cast<uint8_t>(clamp(media, 0.0f, 255.0f));
.....
mediaFiltro.parallel(y).vectorize(x,16); // ← paralelismo + SIMD

```

Figura 22: código paralelo filtro da média

Halide distribui cada linha para um núcleo diferente (`parallel(y)`) e processa 16 pixels de cada vez com instruções SIMD (`vectorize(x, 16)`), garantindo aceleração sem risco de condições de corridas porque cada thread grava linhas distintas.

3.3.1.4. Detecção de bordas

Dois operadores clássicos foram implementados: **Sobel** (gradiente 3 x3 nas direções x e y) e **Roberts** (gradiente 2 x 2). Ambos produzem uma imagem de “magnitude do gradiente” cuja intensidade é maior onde o contraste é mais forte.

A imagem abaixo representa a implementação sequencial em C do **algoritmo Sobel**, onde tudo acontece em uma única thread.

```

for (int c = 0; c < canais; c++) // percorre RGB
    for (int i = 1; i < altura-1; i++) // ignora borda
        for (int j = 1; j < largura-1; j++) {
            float gx = 0, gy = 0; // acumuladores
            for (int dx=-1; dx<=1; dx++) // janela 3x3
                for (int dy=-1; dy<=1; dy++) {
                    int p = imagem.at<Vec3b>(i+dx,j+dy)[c];
                    gx += Gx[dx+1][dy+1] * p; // convolução Gx
                    gy += Gy[dx+1][dy+1] * p; // convolução Gy
                }
            imagemFiltrada.at<Vec3f>(i,j)[c] = sqrt(gx*gx + gy*gy); // magnitude
        }
    }
}

```

Figura 23: código sequencial operador Sobel

Três laços exteriores (canal, linha, coluna) visitam todos os pixels em sequência. Cada gradiente é calculado com duas convoluções manuais dentro da janela 3×3 .

A versão em Halide continua com as diretrizes de otimização `parallel()` e `vectorize(.,)`:

Figura 24: código paralelo operador Sobel

O schedule *parallel(y)* atribui cada linha a um núcleo diferente, e *vectorize(x,16)* reduz 16 operações horizontais a uma instrução SIMD. Como cada thread grava linhas próprias, não há concorrência; Halide insere somente a sincronização final.

3.3.2. Problemas Computacionais de alto desempenho

3.3.2.1. Multiplicação de matrizes

O processo envolve duas etapas principais: o cálculo dos produtos parciais das linhas por colunas e a soma acumulada desses produtos. Nos meus códigos, o paralelismo foi aplicado na etapa de cálculo das linhas da matriz resultado, onde cada thread processa um bloco de linhas de forma independente.

```

for (int i = 0; i < n; i++) {           // percorre linhas de A
    for (int j = 0; j < n; j++) {       // percorre colunas de B ^T
        c[i][j] = 0.0;
        for (int k = 0; k < n; k++) {   // produto interno
            c[i][j] += A[i][k] * B_transposta[j][k];
        }
    }
}

```

Figura 25: código sequencial multiplicação de matrizes

A figura acima representa a versão sequencial do algoritmo em linguagem C, os três laços rodam na mesma thread, o transposto evita cache misses, mas todo o trabalho permanece sequencial. Cada linha A combinada com cada coluna B^T é independente; logo, podemos dividir o laço externo - ou blocos dele - entre vários núcleos ou ainda vetorizá-lo. Abaixo segue a versão paralela de Julia.

```

# Função otimizada usando BLAS para multiplicação de matrizes
function multiplicacao_matriz!(C, A, B)
    mul!(C, A, B) # Usa BLAS otimizado
end

```

Figura 26: código paralelo multiplicação de matrizes

A operação *mul!()* paraleliza a execução distribuindo blocos de linhas e colunas entre os núcleos do processador, utilizando thread-pools otimizados no backend do BLAS. Além disso, dentro de cada bloco, são aplicadas instruções SIMD (Single Instruction Multiple Data) para vetorização, maximizando o uso dos registradores da CPU.

3.3.2.2. Algoritmos de ordenação

O processo de ordenação envolve duas etapas principais: a divisão dos vetores e a ordenação dos subvetores. No **Quicksort**, o paralelismo ocorre após o particionamento, onde as duas partições são processadas de forma independente por diferentes threads. No **Mergesort**, o paralelismo é aplicado desde o início na divisão dos vetores, já que ambas as metades podem ser processadas simultaneamente, pois não há dependência entre elas. Os códigos desenvolvidos exploram justamente essa propriedade.

A imagem abaixo representa a versão sequencial do algoritmo **Quick Sort** escrito em linguagem C.

```

int particionar(int v[], int lo, int hi) {           /* divide e conquista */
    int p = v[hi], i = lo - 1;
    for (int j = lo; j < hi; j++)
        if (v[j] < p) { i++; swap(v[i], v[j]); }
    swap(v[i+1], v[hi]);
    return i + 1;
}

void quicksort(int v[], int lo, int hi) {
    if (lo < hi) {
        int pi = particionar(v, lo, hi);
        quicksort(v, lo, pi-1);          /* chamadas recursivas sequenciais */
        quicksort(v, pi+1, hi);         /* -- tudo em uma única thread -- */
    }
}

```

Figura 27: código sequencial QuickSort

A versão paralela do algoritmo escrito na linguagem Julia é descrito na figura abaixo:

```

pi = particionar!(v, lo, hi)                      # pivô divide o vetor
@threads for side in 1:2                            # cria tasks em paralelo
    if side == 1
        quicksort_parallel!(v, lo, pi-1, depth+1, maxd)
    else
        quicksort_parallel!(v, pi+1, hi, depth+1, maxd)
    end
end

```

Figura 28: código paralelo QuickSort

Depois que o pivô define as duas metades independentes, cada lado vira uma task distribuída pelo agendador de Julia (@threads). Como as duas metades escrevem em regiões distintas do vetor, não há necessidade de locks; a sincronização implícita da diretiva garante que os ramos terminem antes de voltar um nível na pilha.

Utilizamos também a linguagem C para criar a versão tradicional do algoritmo **Mergesort**, conforme figura abaixo:

```

void merge_sort(int v[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        merge_sort(v, l, m);           /* recursão à esquerda */
        merge_sort(v, m+1, r);         /* recursão à direita */
        mesclar(v, l, m, r);          /* funde as metades */
    }
}

```

Figura 29: código sequencial MergeSort

As duas chamadas recursivas são executadas em série e a fusão ocorre após ambas terminarem. Tudo ocorre dentro da mesma thread, de modo que só um núcleo trabalha a cada instante.

A sua versão paralela, escrita em Julia, segue diretivas de sincronismo internos da linguagem:

```

tL = @spawn merge_sort_par!(A, B, l, m, d+1, maxd, thresh)
merge_sort_par!(A, B, m+1, r, d+1, maxd, thresh)
fetch(tL)                      # sincroniza

```

Figura 30: código paralelo MergeSort

`@spawn` delega a metade esquerda a outra thread; a atual ordena a direita. `fetch(tL)` só bloqueia quando precisa, garantindo que ambas terminem antes da fusão. Como cada tarefa trata um segmento distinto, não há escrita nem necessidade de locks.

3.3.2.3. Buscas em grafos (BFS), busca binária com múltiplas chaves

Na **BFS**, exploramos o paralelismo de forma horizontal dentro de cada nível, onde os nós da fronteira atual são processados simultaneamente, embora haja dependência sequencial entre os níveis. Já na **busca binária em lote**, as consultas são completamente independentes, caracterizando um problema embarrassingly parallel, onde cada busca ocorre de forma isolada, permitindo paralelismo máximo sem qualquer dependência entre as tarefas.

O trecho de código abaixo representa a versão sequencial do algoritmo **BFS**:

```

dist[inicio] = 0;
enfileirar(fila, inicio);
...
while (!fila_vazia(fila)) {           // nível k
    int u = desenfileirar(fila);       // percorre vizinhos
    for (int v : adj[u])
        if (dist[v] == -1) {           // ainda não visitado
            dist[v] = dist[u] + 1;      // marca distância
            enfileirar(fila, v);       // coloca na fila de nível k+1
        }
}

```

Figura 31: código sequencial Busca em largura

Todo o laço externo opera em uma única thread: um vértice é retirado, expande-se a sua lista de adjacência e só então o algoritmo avança para o próximo.

A versão paralela do algoritmo, escrito na linguagem OpenMP, segue outra diretiva:

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < tam_fronteira; i++) {
    int u = fronteira[i];
    for (int v : adj[u])
        if (dist[v] == -1 &&
            __sync_bool_compare_and_swap(&dist[v], -1, dist[u] + 1))
            proxima[ __sync_fetch_and_add(&tam_prox, 1) ] = v;
}
```

Figura 32: código paralelo Busca em largura

As linhas da “fronteira” são divididas entre threads. Duas primitivas atômicas garantem correção: *compare_and_swap* marca o vértice como visitado e *fetch_and_add* reserva um slot na próxima fronteira. Não há locks globais e o paralelismo termina quando a fronteira esvazia.

O trecho de código abaixo representa a implementação sequencial do algoritmo de **busca binária em lote**:

```
for (int i = 0; i < NUM_BUSCAS; i++)
resultados[i] = busca_binaria(arr, 0, N-1, consultas[i]);
```

Figura 33: código sequencial Busca com múltiplas chaves

As 100 000 buscas que definimos como parâmetro são feitas em série; cada uma percorre $\log_2 N$ posições e só então a próxima inicia.

A sua versão paralela, escrita com a linguagem OpenMP, é definida da seguinte forma:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < NUM_BUSCAS; i++) {
    resultados[i] = busca_binaria(arr, 0, TAMANHO_ARRAY - 1, consultas[i]);
}
```

Figura 34: código paralelo Busca com múltiplas chaves

static reparte o laço em blocos fixos, reduzindo o overhead de escalonamento. Todos as threads **apenas leem** o vetor ordenado e escrevem em índices exclusivos de *resultados*, logo não há conflito nem necessidade de sincronização adicional.

3.3.2.4. Redução paralela (soma)

Para esse problema o vetor foi dividido em blocos independentes, onde cada thread calcula a soma parcial do seu bloco. No final, essas somas parciais são combinadas para obter o resultado final. Como não há dependências entre os elementos durante a etapa de soma parcial, o paralelismo se torna escalável.

A versão sequencial do algoritmo é descrito na figura abaixo:

```
long long soma = 0;
for (size_t i = 0; i < n; i++)
    soma += vetor[i];           /* 1 x N leituras, 1 acumulador */
```

Figura 35: código sequencial Redução de soma

O laço único percorre o vetor inteiro e acumula cada elemento em *soma*. A variável fica em um só registrador e não muda de thread.

A sua versão paralela escrita em linguagem C com suporte POSIX Threads é exemplificada abaixo:

```
/* 1. divide o vetor em blocos aproximadamente iguais */
size_t bloco = n / num_threads;
...
dados[i].inicio = idx_ini;
dados[i].fim     = idx_ini + bloco + extra; // reparte o resto
pthread_create(&thr[i], NULL, funcao_thread, &dados[i]);

/* 2. cada thread executa: */
void *funcao_thread(void *arg) {
    DadosThread *d = arg;
    long long soma = 0;
    for (size_t k = d->inicio; k < d->fim; k++)
        soma += d->vetor[k];           // soma parcial local, sem disputas
    d->soma_parcial = soma;
}

/* 3. thread principal reúne os resultados */
long long soma_total = 0;
for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
    soma_total += dados[i].soma_parcial; // redução final
}
```

Figura 36: código paralelo Redução de soma

O vetor é dividido em blocos contínuos, com as primeiras threads recebendo um extra se *n* não for múltiplo. Cada thread calcula sua soma local sem necessidade de locks. Ao final, a thread principal acumula os resultados parciais após o *pthread_join*.

3.3.2.5. Monte Carlo para aproximação de π

Para esse problema dividimos a quantidade total de amostras igualmente entre as threads, onde cada uma executa seu processamento de forma totalmente independente, realizando apenas uma etapa final de soma das contagens locais para obter o resultado final.

O trecho de código abaixo representa a versão sequencial do algoritmo:

```
long long contador = 0;
for (long long i = 0; i < N; i++) {           /* gera N pontos          */
    double x = rand()/RAND_MAX*2.0 - 1.0; /* rand ∈ [-1,1]          */
    double y = rand()/RAND_MAX*2.0 - 1.0;
    if (x*x + y*y <= 1.0)                 /* dentro do círculo?      */
        contador++;
}
double pi = 4.0 * contador / N;             /* π ≈ 4·(pontos no círculo)/N */
```

Figura 37: código sequencial Monte carlo

Um único laço gera N amostras, conta quantas caem dentro do círculo e, ao fim, calcula π . Todo o trabalho ocorre em **uma thread**, limitada pela velocidade do gerador `rand()`.

A sua versão paralela escrita em Julia segue a estrutura abaixo:

```
@threads for tid in 1:Threads.nthreads()
    n_local = div(N, nthreads) + (tid ≤ N % nthreads) # divide amostras
    rng     = MersenneTwister(tid + 1234)              # RNG privado
    cont[tid] = monte_carlo_pi_thread(n_local, rng)    # contagem local
end
π = 4.0 * sum(cont) / N                                # redução final
```

Figura 38: código paralelo Monte carlo

`@threads` reparte as amostras quase igualmente; cada thread usa um gerador próprio, soma pontos dentro do círculo em uma variável local (`cont[tid]`) e, no fim, a thread principal apenas agrupa o vetor `cont`. Não há locks – o problema é intrinsecamente independente.

4. EXPERIMENTOS E RESULTADOS

4.1. Comparações de Desempenho em Processamento de Imagens

Os experimentos realizados para avaliar o desempenho das técnicas de **processamento de imagens** demonstraram ganhos significativos com o uso de programação paralela, especialmente quando aplicadas em arquiteturas multicore modernas, neste caso em duas máquinas distintas: a primeira com capacidade de 1 a 4

threads com CPU i5-7200U, a segunda com capacidade de 1 a 24 threads com CPU i7-137004.

Foram analisadas três categorias principais: **transformações radiométricas**, **filtragens espaciais** e **detecção de bordas**, todas comparadas nas linguagens Python, C e Halide.

Excepcionalmente, para a técnica de **Limiarização de Otsu**, os testes foram realizados em três máquinas com capacidades distintas: a primeira com processador **Intel Core i7-13700 (16 núcleos/24 threads)**, a segunda **i5-12500H (8 núcleos/16 threads)** e por último **i5-9500 (6 núcleos/6 threads)**. Diferente das técnicas citadas acima onde os testes foram realizados em apenas duas máquinas.

Isso ocorre devido a técnica descrita ter sido testada e implementada no segundo ano de projeto quando se expandiu ainda mais as arquiteturas selecionadas para testes. As linguagens utilizadas para implementar foram **C para tradicional, OpenMP, Julia e Halide para abordagens paralelas**. A tabela abaixo representa os dados coletados durante os experimentos, todos em segundos.

Arquitetura	C	C+Threads	Julia	OpenMP	Halide
6 threads	262.9244	136.9614	359.0624	168.2069	0.6176
16 threads	145.8125	102.8712	346.0395	101.9961	0.5826
24 threads	108.8022	106.5027	203.7626	80.2740	0.1182

Tabela 4: Resultados Limiarização de Otsu

Na arquitetura com 6 threads, Halide foi a abordagem mais eficiente, com **0,61 s**, superando todas as demais. OpenMP e C com Threads apresentaram **168,20 s** e **136,96 s**, respectivamente, enquanto o C sequencial ficou com **262,92 s**. Julia foi a mais lenta nesse cenário, com **359,06 s**.

Com 16 threads, Halide manteve o menor tempo (**0,58 s**), seguido por OpenMP (**101,99 s**) e C com Threads (**102,87 s**). O tempo do C sequencial caiu para **145,81 s**, e Julia para **346,03 s**, mas ambas ainda mantiveram desempenho inferior.

Na arquitetura com 24 threads, Halide foi ainda mais rápido, com apenas **0,11 s**, enquanto OpenMP obteve **80,27 s**, e C com Threads **106,50 s**. C sequencial registrou **108,80 s**, e Julia reduziu para **203,76 s**, ainda permanecendo como a mais lenta e não adequada para a técnica testada.

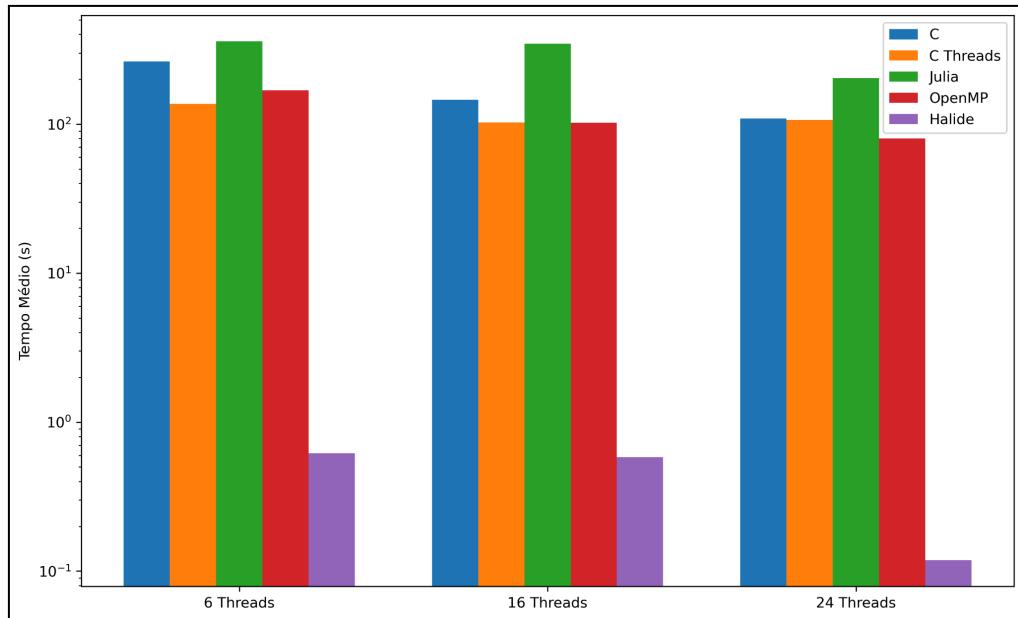


Figura 39: Resultados Limiarização de Otsu

A Figura acima ilustra claramente o domínio de Halide, com tempos visivelmente inferiores nas três arquiteturas, evidenciado o aumento de desempenho de acordo com o aumento do número de threads. As demais abordagens mantêm diferenças pequenas entre si, com destaque negativo para Julia, que obteve os maiores tempos em todas as configurações.

Na categoria de **transformadas radiométricas**, foram avaliadas três técnicas: **expansão linear de contraste, logaritmo e dente de serra**. Os resultados obtidos estão sintetizados nas tabelas a seguir, separadas por configuração de hardware e levando em consideração a média das execuções.

Na tabela abaixo é apresentado os resultados obtidos na máquina com até **4 threads** (**CPU Intel Core i5-7200U**), utilizados como cenário de base para avaliar ganhos mesmo em ambientes com paralelismo mais limitado.

Técnica	Python (ms)	C (ms)	Halide (ms)
Expansão Linear	54202.6059	272.8832	185.4667
Logaritmo	40204.0658	421.8791	211.5000
Dente de Serra	119273.2424	260.5805	189.3667

Tabela 5: Resultados Transformadas Radiométricas 4 threads

Na técnica de dente de serra, por exemplo, Halide foi **628 vezes mais rápido que Python e 1 vez mais rápido que C**. Já na transformação logarítmica, Halide superou Python em **190 vezes**, enquanto C foi 95 vezes mais rápido que Python. O gráfico abaixo mostra a relação entre as linguagens e configuração de hardware para as transformadas.

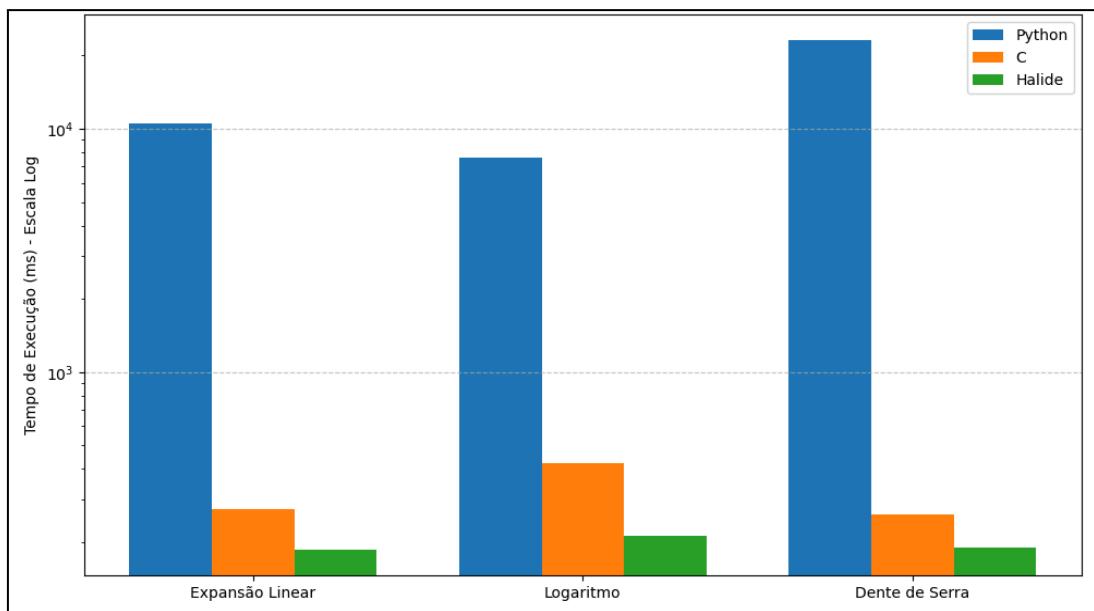


Figura 40: Resultados Transformadas Radiométricas 4 threads

Observando o gráfico, a linguagem Halide, opção paralela, se destaca mesmo em um ambiente com poucos núcleos. Os tempos de execução para as três técnicas foram consistentemente menores em Halide quando comparados ao C e drasticamente inferiores aos obtidos em Python, que ultrapassaram os 10 segundos em todas as execuções.

Na arquitetura de **24 threads (CPU i7-137004)**, os ganhos se tornaram ainda mais expressivos. A tabela exibe os tempos médios de execução para as mesmas técnicas, agora em uma máquina com arquitetura híbrida e suporte a múltiplas threads simultâneas por núcleo.

Técnica	Python (ms)	C (ms)	Halide (ms)
Expansão Linear	10540.8364	108.4604	52.2667
Logaritmo	7649.6159	154.5248	57.9667
Dente de Serra	23112.5866	121.7629	54.6000

Tabela 6: Resultados Transformadas Radiométricas 24 threads

Com essa nova configuração, Halide foi capaz de reduzir ainda mais os tempos: na técnica de dente de serra, ele foi **423 vezes mais rápido que Python e 2 vezes mais rápido que C**. Já na expansão linear, os ganhos foram de **200 vezes sobre Python e 2 vezes sobre C**. O gráfico abaixo exemplifica bem os dados citados acima:

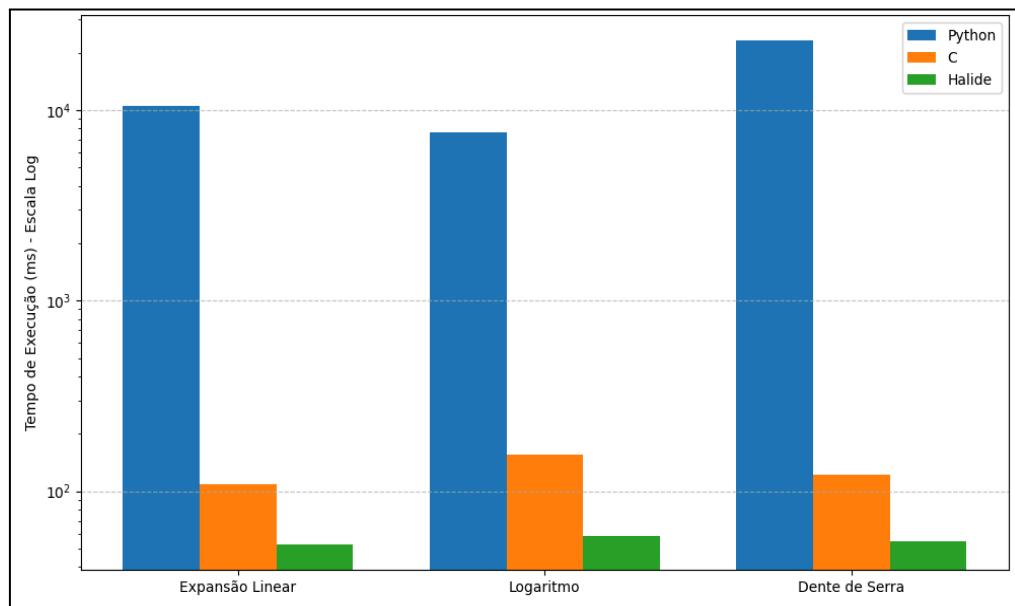


Figura 41: Resultados Transformadas Radiométricas 24 threads

A escalabilidade de Halide é evidente: a linguagem não só mantém sua vantagem mesmo com maior paralelismo, como a amplifica.

Na categoria de **filtragens espaciais**, foram analisadas três técnicas comuns para suavização e remoção de ruídos em imagens: **filtro da média, mediana e K-vizinhos mais próximos**. A tabela abaixo apresenta os dados coletados na máquina com **4 threads (Intel Core i5-7200U)**.

Técnica	Python (ms)	C (ms)	Halide (ms)
Média	450812.6250	5138.1128	442.1333
Mediana	1105394.0205	98179.7097	663.8667
K-vizinhos	918534.6280	225874.5325	1084.3000

Tabela 7: Resultados Filtragens Espaciais 4 threads

Na máquina descrita, os resultados mostram que as versões em Python apresentaram os maiores tempos em todas as técnicas, com destaque negativo para o filtro da mediana. Para a versão em C o tempo foi reduzido, mas ainda apresentou valores elevados nas filtragens não lineares, especialmente nos k vizinhos.

Halide obteve os melhores tempos nas três técnicas, com a filtragem da média executando em apenas 442 ms e a da mediana em 663 ms, representando ganhos de mais de **1000 vezes em relação ao Python** e até **147 vezes em relação ao C**. Na filtragem dos **k vizinhos mais próximos**, Halide também se destacou, com **1.084 ms**, contra **225.874 ms** da versão em C, confirmando a superioridade da versão paralela nas três técnicas analisadas.

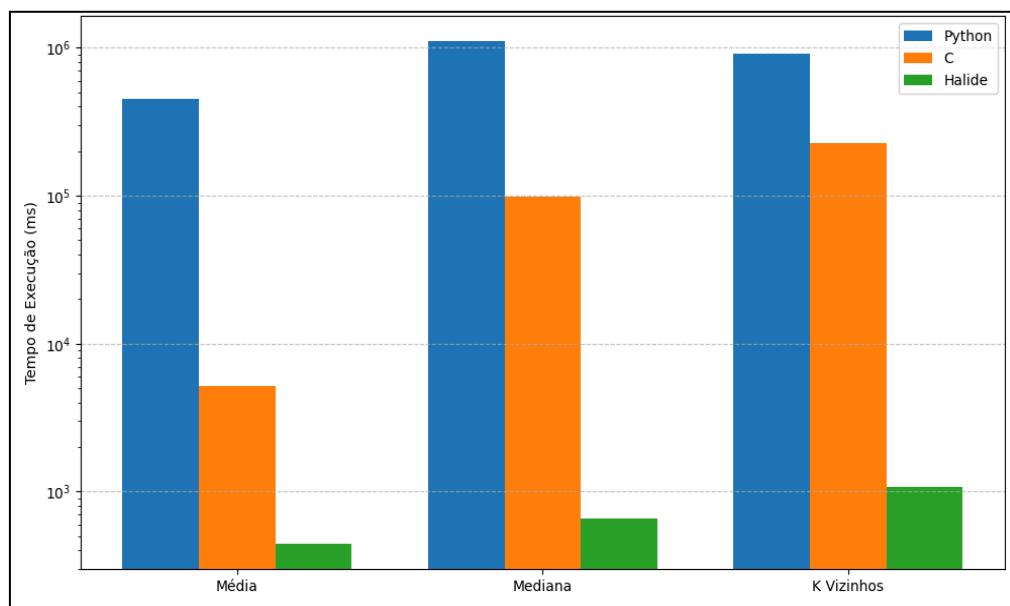


Figura 42: Resultados Filtragens Espaciais 4 threads

Na Figura acima, os dados confirmam a magnitude das diferenças de desempenho entre as linguagens e abordagens adotadas. Os tempos de Python ocupam uma ordem de grandeza maior. Halide novamente se destaca pela consistência na redução dos tempos,

especialmente nas técnicas lineares como a média, a versão em C apresenta ganhos inferiores a Python, mas superior a Halide.

A tabela abaixo apresenta os dados coletados para as mesmas técnicas, no entanto na arquitetura que suporta **24 threads lógicas (Intel Core i7-13700)**.

Técnica	Python (ms)	C (ms)	Halide (ms)
Média	89158.4204	1175.0984	108.4333
Mediana	197033.2473	20485.0092	167.2000
K-vizinhos	175001.3283	42856.6879	256.9000

Tabela 8: Resultados Filtragens Espaciais 24 threads

Com maior capacidade de paralelismo, a **máquina de 24 threads** permitiu reduções expressivas nos tempos de execução, principalmente nas versões em Python e C, embora ainda permaneçam significativamente mais lentas que Halide.

Na filtragem da média, o tempo foi reduzido de 450 mil para 89 mil ms em Python e para 1.175 ms em C, enquanto Halide executou a mesma operação em apenas 108 ms, sendo **10 vezes mais rápido que C** e mais de **800 vezes mais rápido que Python**.

A mediana, também teve excelente desempenho com Halide (167 ms), superando as demais linguagens com **ganhos superiores a 100 vezes**. Na filtragem dos k vizinhos, Halide manteve desempenho superior ao Python e ao C.

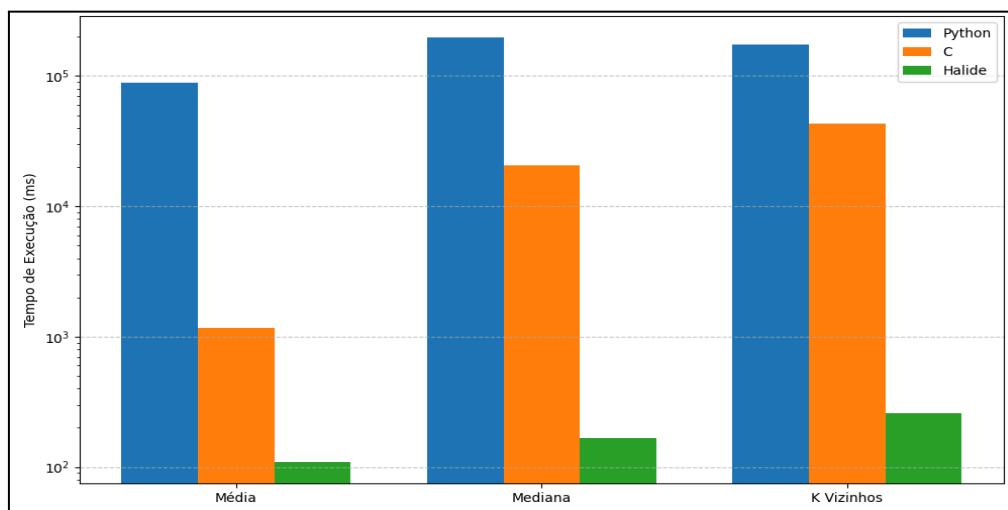


Figura 43: Resultados Filtragens Espaciais 24 threads

A Figura acima evidencia o impacto da arquitetura paralela mais robusta na execução dos algoritmos. Os tempos de Halide permanecem muito inferiores em relação às outras abordagens. As melhorias foram ainda mais expressivas nas técnicas da média e da mediana, consolidando Halide como a linguagem de melhor desempenho para este conjunto de filtros espaciais.

Na categoria de **Detecção de Bordas**, foram avaliadas duas abordagens clássicas: o **operador de Sobel** e o **operador de Roberts**. A seguir, são apresentados os tempos médios de execução.

Primeiramente, a tabela contendo os dados para a máquina com **4 threads lógicas (Intel Core i5-7200U)**:

Técnica	Python (ms)	C (ms)	Halide (ms)
Sobel	786337.9029	5574.5200	1230.2667
Roberts	816129.8633	4750.9302	1006.5000

Tabela 9: Resultados Detecção de Bordas 4 threads

Na máquina descrita, os algoritmos de detecção de bordas apresentaram tempos elevados nas versões implementadas em Python, com valores acima de 780 mil milissegundos para ambas as técnicas. As versões em C já demonstraram avanços significativos, especialmente com o operador de Roberts, que executou 4.750 ms. No entanto, mais uma vez, a linguagem **Halide obteve o melhor desempenho absoluto**, com 1.230 ms para Sobel e 1.006 ms para Roberts, evidenciando **ganhos superiores a 600 vezes em relação ao Python** e aproximadamente **4 vezes mais rápido que C**.

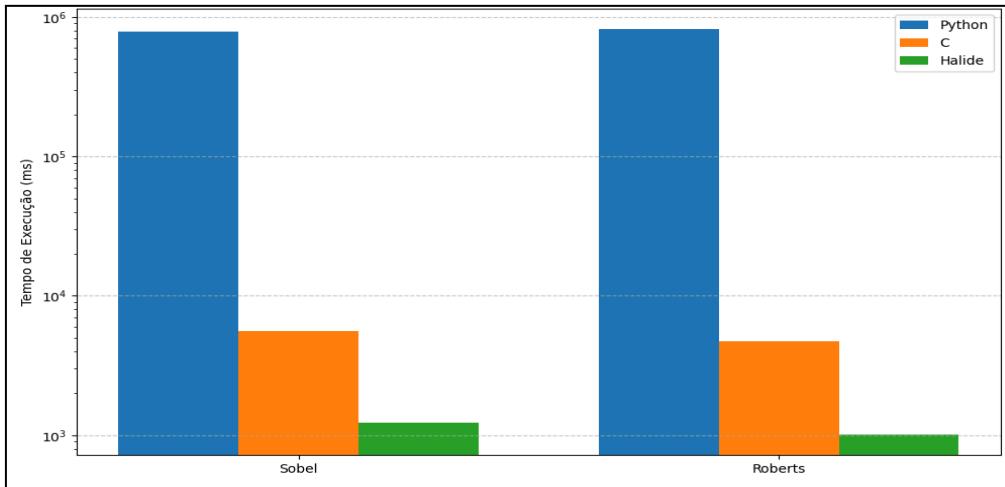


Figura 44: Resultados Detecção de Bordas 4 threads

Observando o gráfico acima, Python mantém tempos de execução extremamente altos, enquanto C oferece melhorias, mas é **Halide quem se destaca**, mantendo o tempo em torno de **mil milissegundos**, mesmo em uma máquina com capacidade paralela modesta. A diferença entre as técnicas (Sobel e Roberts) foi pequena, indicando que ambas se beneficiam de forma semelhante do paralelismo oferecido pelo Halide.

Em seguida, é destacado a tabela contendo os dados coletados a partir da máquina com maior capacidade de paralelismo, precisamente **24 threads lógicas (Intel Core i7-13700)**.

Técnica	Python (ms)	C (ms)	Halide (ms)
Sobel	163969.2557	1367.5888	260.1667
Roberts	161012.9039	1048.4216	234.8667

Tabela 10: Resultados Detecção de Bordas 24 threads

Os tempos de execução caíram em todas as linguagens, com destaque para o Python, que reduziu os tempos de mais de 780 mil para aproximadamente 160 mil ms, e o C, que caiu para pouco mais de mil milissegundos. Ainda assim, **Halide permaneceu como a solução mais eficiente**, com tempos de 260 ms (Sobel) e 234 ms (Roberts), o que representou uma melhoria de até **685 vezes em relação ao Python** e cerca de **4 vezes em relação ao C**.

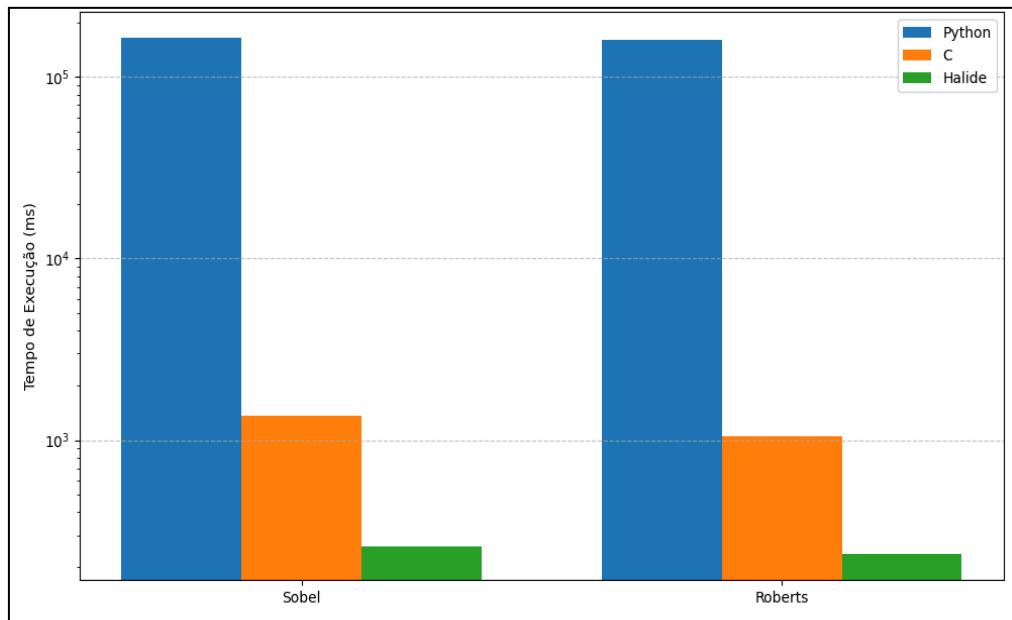


Figura 45: Resultados Detecção de Bordas 24 threads

A **Figura 45** demonstra claramente a capacidade de escalabilidade da linguagem Halide. Enquanto Python e C também se beneficiam de acordo com o avanço das arquiteturas, é o **Halide que atinge os menores tempos absolutos**, com desempenho equilibrado entre Sobel e Roberts. A consistência nos ganhos reforça a eficiência da linguagem para operações de gradiente, típicas da detecção de bordas, mesmo sob diferentes tamanhos de imagem e arquiteturas de hardware.

4.2. Comparações dos problemas computacionais de alto desempenho

Após a apresentação dos resultados referentes ao processamento de imagens, esta seção aborda o **desempenho das demais aplicações computacionais analisadas**, classificadas como problemas computacionais de alto desempenho. Esses problemas envolvem algoritmos que, embora não sejam específicos do processamento de imagens, são amplamente utilizados em diversos domínios computacionais e científicos, como álgebra linear, ordenações, buscas e simulações numéricas.

Para tanto, vale salientar que os algoritmos foram desenvolvidos na **Linguagem C para versão tradicional, OpenMP, Halide, C+Threads e Julia** para as **versões paralelas** e testados em **configurações diferentes de hardware** para verificar a escalabilidade, eficiência e tempo de execução de acordo com o aumento da quantidade de núcleos. A tabela abaixo exemplifica as características de cada uma.

Núcleos/Threads	Processador	Arquitetura	Tipo de núcleo
16/24	Intel Core i7-13700	x86_64	Híbrido (P-cores + E-cores)
8/16	Intel Core i5-12500H	x86_64	Híbrido (P-cores + E-cores)
6/6	Intel Core i5-9500	x86_64	P-cores (Tradicional)

Tabela 11: Configurações de Hardware

Nesta subseção, iniciamos a análise com a **multiplicação de matrizes**, um dos problemas mais clássicos e relevantes para avaliação de desempenho em ambientes paralelos.

Neste trabalho, foram utilizados tamanhos crescentes de matrizes quadradas (de 1.000×1.000 até 10.000×10.000), permitindo avaliar o impacto da escalabilidade em diferentes arquiteturas. Abaixo segue as médias dos tempos de execução em segundos coletados durante os experimentos.

Arquitetura	C	C+Threads	Julia	OpenMP	Halide
6 threads	960.9064	137.4125	4.3916	34.3318	17.7840
16 threads	753.2301	29.6916	3.3176	28.2913	10.4786
24 threads	720.0377	15.2394	2.6498	14.7701	7.4783

Tabela 12: Resultados multiplicação de matrizes

A análise dos dados mostra que a **linguagem C** obteve os maiores tempos de execução em todas as arquiteturas, com média de **960,90 s** na máquina com **6 threads**, reduzindo levemente para **720,03 s** com **24 threads**, uma queda modesta devido à ausência de paralelismo. Em contrapartida, todas as abordagens paralelas apresentaram ganhos significativos, especialmente **Julia**, que se destacou como a mais eficiente em todas as arquiteturas.

Na máquina com **6 threads**, Julia foi **218 vezes mais rápida que C**. O mesmo comportamento foi mantido nas máquinas com **16 threads (227 vezes)** e **24 threads (271 vezes)**. Esse desempenho elevado está associado ao uso da biblioteca **BLAS** (Basic Linear Algebra Subprograms), altamente otimizada com suporte a paralelismo multi-thread, vetorização SIMD.

A linguagem **Halide** também apresentou bom desempenho, ficando em segundo lugar nas três arquiteturas, com **tempo médio de 17,78 s na máquina com 6 threads**, reduzindo para **7,48 s com 24 threads**. Isso representa uma **aceleração de até 96 vezes em relação ao C**.

OpenMP e C com threads mostraram desempenho intermediário. OpenMP foi mais eficiente nas arquiteturas maiores, reduzindo o tempo de 34,33 s (6 threads) para 14,77 s (24 threads), com ganhos que chegam a **48 vezes em relação ao C sequencial**. C com threads, também apresentou ganhos importantes (de **137,41 s para 15,24 s**).

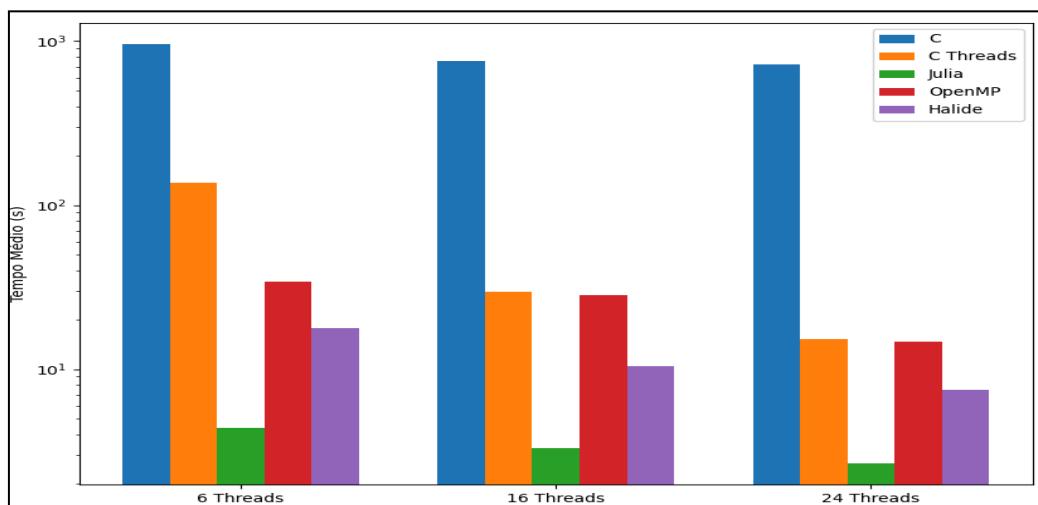


Figura 46: Resultados multiplicação de matrizes

A Figura acima mostra que, independentemente da arquitetura, **Julia manteve-se como a linguagem mais eficiente**. O gráfico também confirma a escalabilidade das técnicas paralelas: todas as abordagens diminuíram seus tempos à medida que a arquitetura se tornou mais robusta, sendo a máquina com 24 threads a que obteve os melhores resultados.

Nesta etapa, são analisados os **algoritmos de ordenação Quick Sort e Merge Sort**, ambos utilizados em aplicações que exigem processamento eficiente de grandes volumes de dados. Abaixo são apresentados os tempos médios de execução em segundos dos algoritmos por arquitetura, variando os vetores em até 100 milhões de elementos.

Arquitetura	Algoritmo	C	C+Threads	Julia	OpenMP	Halide
6 threads	Quick	41.6062	14.3771	1.5433	3.6310	316.1663
	Merge	30.1933	12.0214	2.9246	2.0838	95.3549
16 threads	Quick	17.0876	7.7808	1.1832	2.6287	178.6420
	Merge	16.5298	5.2898	1.4747	1.7659	55.6585
24 threads	Quick	12.0266	5.7880	0.6326	1.0934	18.7133
	Merge	9.4988	5.0792	0.3874	0.7155	11.7681

Tabela 13: Resultados algoritmos de ordenação

Os dados revelam que **Julia obteve o melhor desempenho em todas as configurações e algoritmos**, mantendo tempos inferiores aos demais. Na arquitetura com **24 threads**, Julia executou o Quick Sort em apenas **0,63 segundos** e o Merge Sort em **0,38 segundos**, representando um ganho de aproximadamente **19 vezes e 24 vezes**, respectivamente, em relação à versão sequencial em C.

OpenMP e C com Threads também apresentaram ganhos relevantes. No Quick Sort com 24 threads, OpenMP atingiu **1,09 s** e C com Threads **5,78 s**, o que representa um ganho de até **11 vezes sobre o C sequencial**. No Merge Sort, OpenMP manteve vantagem sobre C Threads, com **0,71 s** contra **5,07 s**.

Halide, por outro lado, se mostrou a **menos adequada para algoritmos de ordenação**, principalmente no Quick Sort. Na arquitetura de 6 threads, levou **316,16 s**, enquanto o C sequencial executou **41,60 s**. Isso ocorre porque o Quicksort exige controle de fluxo recursivo dinâmico, o que está fora do modelo de execução otimizado de Halide.

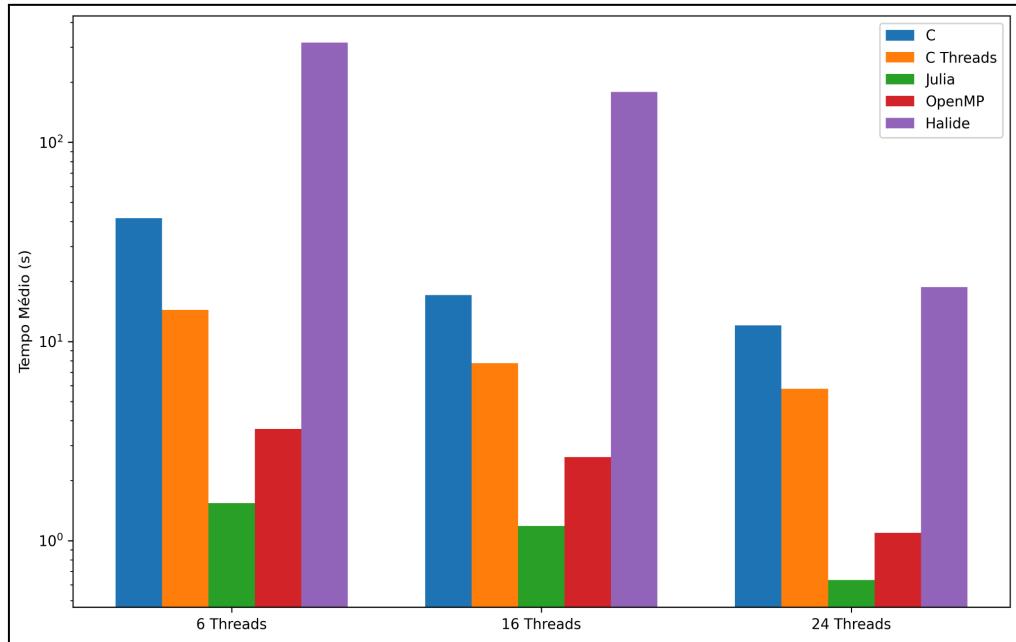


Figura 47: Resultados QuickSort

A figura acima deixa claro em relação ao **QuickSort**: as barras de Julia são visivelmente menores, mesmo com apenas 6 threads, seguidas por OpenMP e C com Threads. Halide, ao contrário, ocupa sempre o topo da escala logarítmica, demonstrando que seu modelo não é indicado para este tipo de algoritmo.

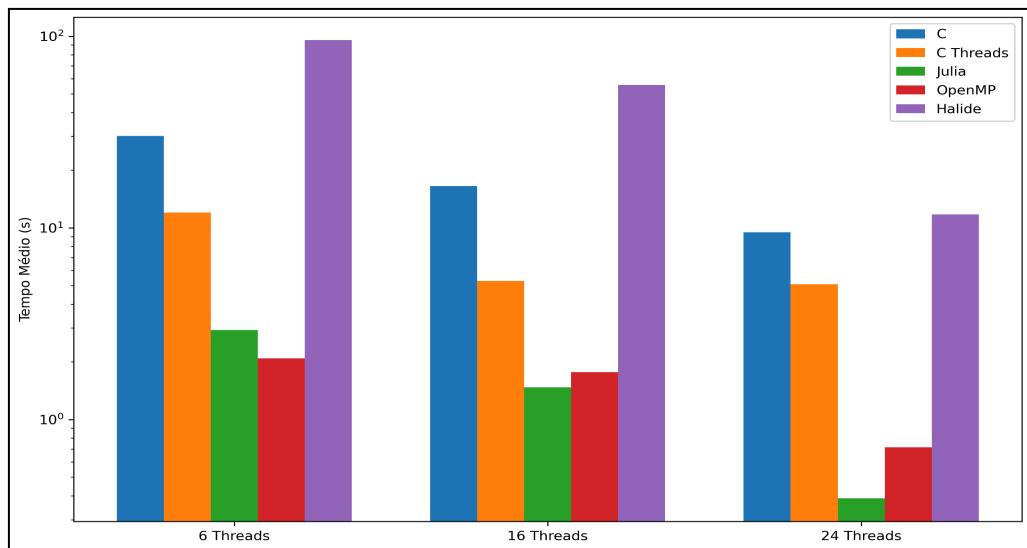


Figura 48: Resultados MergeSort

Já a **Figura 48 (Merge Sort)** mostra uma maior regularidade entre as abordagens paralelas, com destaque para a escalabilidade de Julia e OpenMP — o que confirma que o

Mergesort é estruturalmente mais amigável ao paralelismo, por permitir divisão balanceada desde as chamadas recursivas iniciais.

Nesta subseção são avaliadas duas operações de busca distintas: a **Busca em Largura (BFS)**, aplicada sobre grafos, e a **Busca Binária aplicada a múltiplas chaves em um vetor ordenado**.

As buscas foram realizadas sobre estruturas de dados suficientemente grandes para simular cenários realistas de uso, especialmente voltados ao processamento massivo de dados. Abaixo seguem os dados coletados em segundos nas diferentes configurações de hardware.

Arquitetura	C	C+Threads	Julia	OpenMP	Halide
6 threads	2.278725	0.914257	3.192517	0.836152	109.383144
16 threads	1.737764	0.616510	1.081269	0.663264	49.739951
24 threads	1.324812	0.281340	0.776787	0.345375	23.6479

Tabela 14: Resultados Busca em largura

Os resultados da BFS demonstram um **comportamento sensível à arquitetura e à linguagem escolhida**. Na máquina com 6 threads, a versão sequencial em C executou em **2,27 s**, enquanto a abordagem com **C com Threads** reduziu esse tempo para **0,91 s**, representando um ganho de **2,5 vezes**.

OpenMP teve desempenho ainda melhor (**0,83 s**), mas a versão com **Julia** foi inferior, com **3,19 s**, devido ao overhead no gerenciamento de fronteiras e junção de dados. **Halide**, por sua vez, teve o pior desempenho, com **109,38 s**, o que evidencia sua **ineficiência estrutural para algoritmos baseados em estado e controle iterativo**, como é o caso da BFS.

Na arquitetura com 24 threads, os ganhos se acentuam. A BFS com **C com Threads** foi executada em apenas **0,28 s**, sendo quase **5 vezes mais rápida que o C sequencial (1,32 s)**. **OpenMP** seguiu de perto, com **0,34 s**, enquanto Julia obteve **0,77 s**, ainda atrás das demais abordagens paralelas. Halide, mesmo com melhora, ainda registrou **23,64 s**, mantendo-se como a abordagem menos adequada para esse tipo de algoritmo.

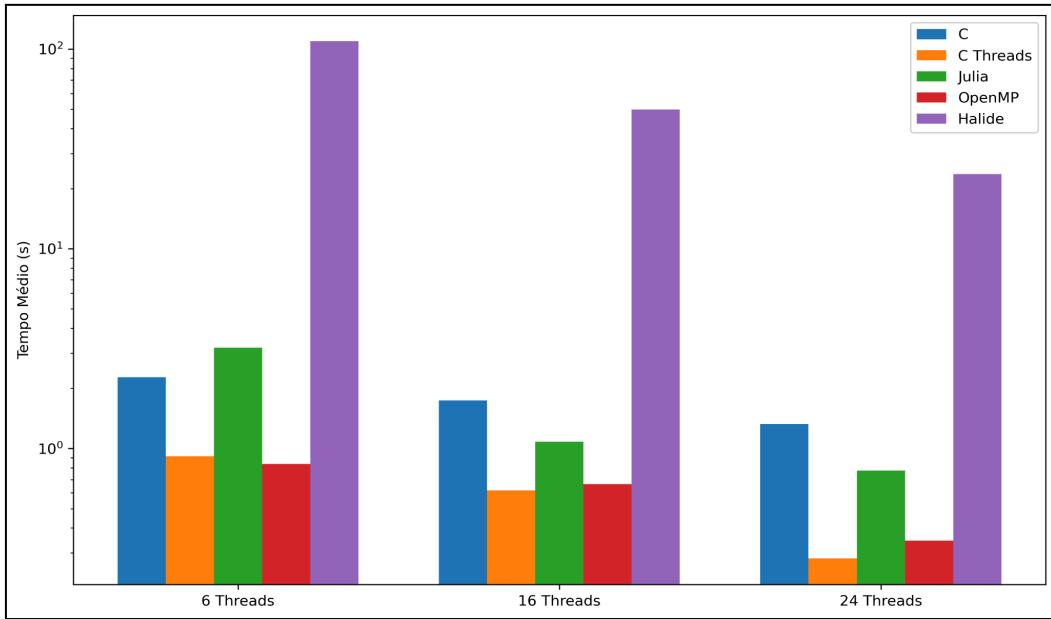


Figura 49: Resultados Busca em largura

A figura acima ilustra as diferenças observadas: **C com Threads e OpenMP apresentam as barras mais baixas**, demonstrando grande eficiência. Julia mantém desempenho intermediário, enquanto Halide aparece como o mais lento, refletindo sua limitação com estruturas iterativas e controle dependente do grafo.

Na operação de **busca binária**, cada linguagem foi testada para **múltiplas chaves simultaneamente**, cenário que permite o uso eficiente de paralelismo por divisão de tarefas entre threads. O padrão de desempenho aqui é mais favorável para todas as abordagens, observadas na tabela abaixo.

Arquitetura	C	C+Threads	Julia	OpenMP	Halide
6 threads	0.142690	0.009940	0.062802	0.031693	0.098301
16 threads	0.048473	0.000553	0.005025	0.015981	0.054386
24 threads	0.049737	0.000222	0.003040	0.005713	0.043372

Tabela 15: Resultados Busca com múltiplas chaves

Na máquina com 6 threads, **C com Threads** foi a linguagem mais rápida (**0,0099 s**), enquanto OpenMP obteve **0,031 s** e Julia **0,062 s**. Halide superou o C sequencial, executando em **0,098 s**, contra **0,14 s** de C. Isso demonstra que, apesar de sua limitação

para grafos, Halide consegue aplicar vetorização e paralelismo eficiente quando o padrão de acesso é previsível e orientado por dados fixos.

Com 24 threads, os ganhos foram ainda maiores. **C com Threads** reduziu o tempo para apenas **0,00022 s**, seguido por **OpenMP (0,0057 s)** e **Julia (0,0030 s)**. **Halide** executou a operação em **0,043 s**, enquanto o C sequencial permaneceu em **0,049 s**. A Figura abaixo mostra que todas as abordagens paralelas superaram o C, com **C com Threads e Julia se destacando visivelmente**.

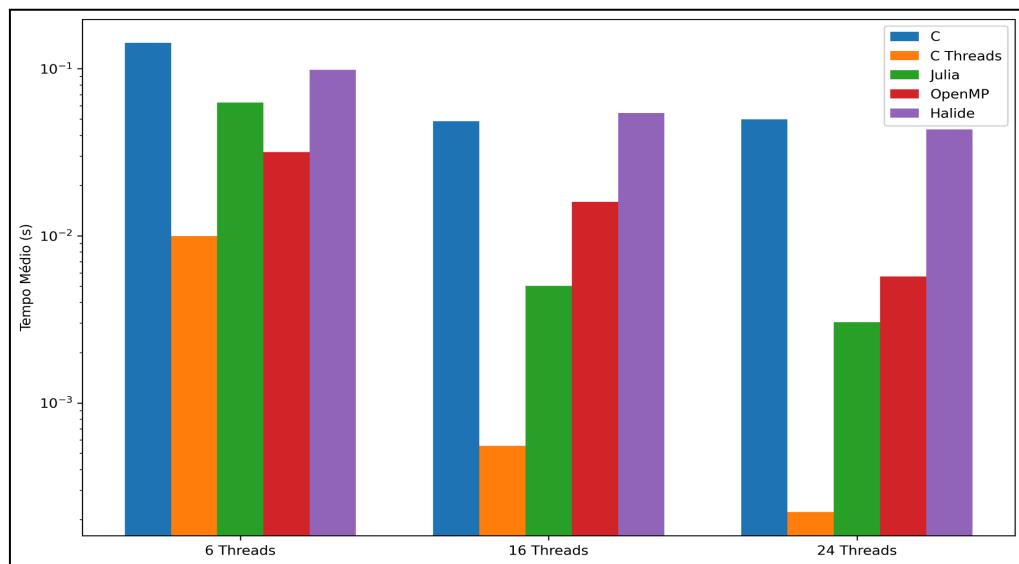


Figura 50: Resultados Busca com múltiplas chaves

Este cenário revela que **a busca binária é altamente paralelizável quando aplicada sobre múltiplas chaves**. A divisão do vetor de buscas entre as threads resulta em operações independentes, sem necessidade de sincronização entre tarefas. Esse tipo de paralelismo é ideal para C Threads, OpenMP e Julia, e até mesmo Halide consegue desempenho competitivo, dada a previsibilidade do domínio de execução.

A próxima operação analisada é a **redução** que consiste na agregação de valores de um vetor em um único resultado, comumente utilizando somas, produtos ou outras operações associativas. Neste experimento, foi aplicada a **soma de elementos de um vetor**, abaixo segue os dados coletados, em segundos.

Arquitetura	C	C+Threads	Julia	OpenMP	Halide
6 threads	0.190071	0.012166	0.024399	0.011762	0.054981
16 threads	0.111959	0.011613	0.023218	0.011762	0.053212
24 threads	0.105455	0.009862	0.019900	0.011148	0.040764

Tabela 16: Resultados Redução de soma

A versão **sequencial em C** apresentou os maiores tempos de execução em todas as arquiteturas. Na máquina com 6 threads, o tempo foi de **0,190 s**, enquanto na de 24 threads foi reduzido para **0,105 s**.

Por outro lado, todas as abordagens paralelas apresentaram **melhor desempenho**, com **OpenMP e C com Threads se destacando** como as mais eficientes. Com 24 threads, OpenMP alcançou **0,0111 s**, enquanto C com Threads registrou **0,0098 s**, o menor tempo da análise, o que representa um ganho de aproximadamente **10 vezes em relação ao C sequencial**.

Julia apresentou desempenho consistente, com tempos entre **0,0199 s e 0,0243 s**, demonstrando bom aproveitamento da arquitetura, mas ainda inferior às abordagens com OpenMP e Threads manuais. Já **Halide**, embora mais lenta que os demais paralelos, foi superior ao C sequencial, executando em **0,0407 s** com 24 threads.

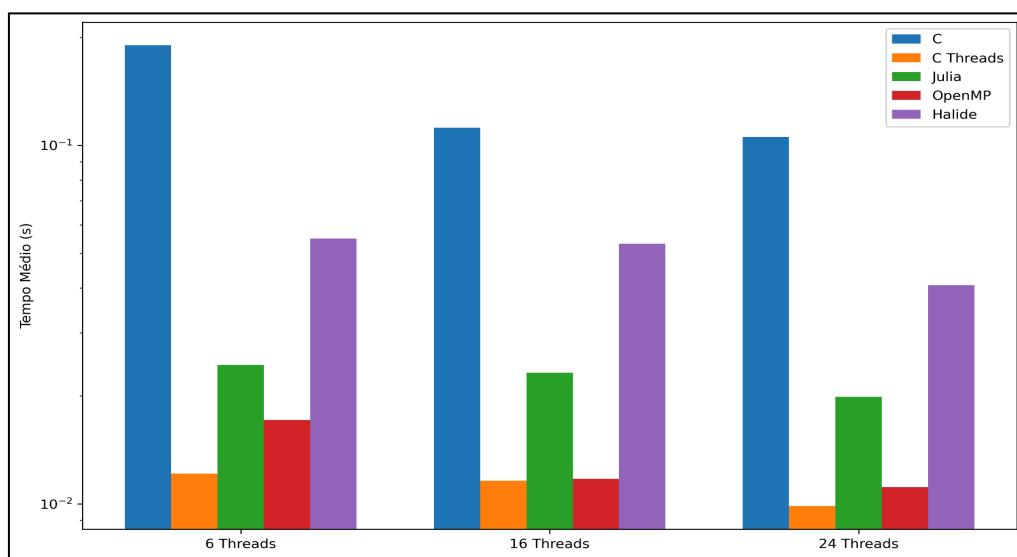


Figura 51: Resultados Redução de soma

A **Figura 51** reforça visualmente que as barras de OpenMP e C Threads são as mais baixas em todas as arquiteturas. A linguagem Julia aparece logo acima, com Halide mantendo desempenho intermediário e C sequencial se destacando negativamente.

A pequena diferença entre OpenMP e C Threads evidencia que **ambas foram capazes de explorar eficientemente o paralelismo por blocos**, sendo a sincronização mínima e a estrutura do problema ideal para a paralelização por dados.

Por fim, a última técnica analisada é o método de **Monte Carlo**, uma técnica probabilística amplamente utilizada para resolver problemas numéricos, especialmente aqueles que envolvem integrações, simulações físicas e aproximações estatísticas.

Neste experimento, o algoritmo foi utilizado para **aproximar o valor de π** , a partir da geração de milhões de pontos aleatórios (x, y) e da contagem de quantos caem dentro de um círculo de raio 1. Abaixo seguem os dados na grandeza de segundos.

Arquitetura	C	C+Threads	Julia	OpenMP	Halide
6 threads	2.466769	0.244828	0.033378	0.080879	41.456450
16 threads	1.521815	0.193949	0.025570	0.081161	8.837282
24 threads	1.410556	0.127606	0.012815	0.038465	7.641219

Tabela 17: Resultados Monte carlo

Na arquitetura com 6 threads, Julia foi a linguagem mais eficiente, com tempo médio de **0,033 s**, superando todas as demais abordagens paralelas. OpenMP e C com Threads registraram **0,080 s** e **0,244 s**, respectivamente. O C sequencial apresentou o pior desempenho entre os paralelos com **2,46 s**, mas abaixo de Halide que teve o maior tempo absoluto: **41,45 s**.

Com 16 threads, Julia manteve o melhor desempenho (**0,025 s**), seguida por OpenMP (**0,081 s**) e C com Threads (**0,193 s**). O C sequencial ainda permaneceu lento, com **1,52 s**, e Halide reduziu o tempo para **8,83 s**, mantendo-se como a abordagem menos eficiente.

Na arquitetura com 24 threads, os menores tempos permaneceram com Julia (**0,012 s**) e OpenMP (**0,038 s**), enquanto Halide ainda apresentou desempenho inferior até mesmo

em relação ao tradicional. C com Threads atingiu **0,127 s**, e o C sequencial, **1,41 s**, permanecendo lento.

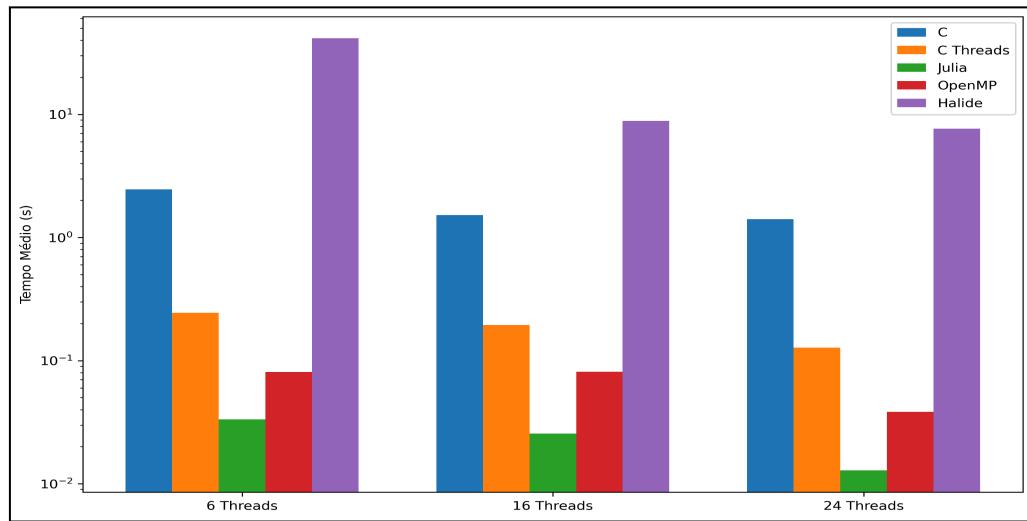


Figura 52: Resultados Resultados Monte carlo

A **Figura 52** destaca Julia como a linguagem mais rápida em todas as arquiteturas. OpenMP e C com Threads apresentam tempos semelhantes e consistentes. Halide mantém os maiores tempos em todos os casos, sendo superado até pelo C sequencial.

4.3. Discussão dos Resultados

Os resultados obtidos nas comparações experimentais revelam de forma clara como a combinação entre paralelismo, arquitetura de hardware e linguagem de implementação influencia decisivamente no desempenho de algoritmos computacionais, tanto em processamento de imagens quanto em aplicações de alto desempenho. Esses resultados confirmam padrões de aceleração já descritos na literatura de computação paralela, especialmente para tarefas altamente paralelizáveis (GRAMA et al., 2003).

Inicialmente, observou-se que a escalabilidade dos algoritmos paralelos é fortemente impactada pelo número de núcleos disponíveis na máquina. As arquiteturas mais modernas, como a do Intel Core i7-13700 com 24 threads, se destacaram em praticamente todos os cenários analisados. Essa arquitetura híbrida, composta por núcleos de performance (P-cores) e eficiência (E-cores), foi capaz de alavancar o desempenho das implementações paralelas com ganhos expressivos.

No campo das linguagens e frameworks, Julia demonstrou ser altamente eficaz em algoritmos massivamente paralelizáveis. Em todos os problemas analisados a linguagem se beneficiou do aumento de threads disponíveis. Isso se explica pelo fato de Julia delegar

tarefas pesadas às bibliotecas BLAS, que utilizam paralelismo multinível com vetorização SIMD e agendamento dinâmico, explorando eficientemente desde poucos até muitos núcleos. Seu desempenho melhora de forma quase proporcional à capacidade de paralelismo do sistema.

Por outro lado, o OpenMP apresentou resultados interessantes que revelam seu papel contextual. Embora tenha sido superado por Julia em arquiteturas mais modernas, sua eficácia é amplamente observada em sistemas com menor capacidade de paralelismo, como a máquina com 6 threads. Esse comportamento é coerente com o propósito original do OpenMP: quando criado, ainda não havia arquiteturas com dezenas de núcleos, e suas diretivas foram desenhadas para explorar paralelismo em contextos mais limitados.

C com threads mostrou-se consistente em problemas que se beneficiam de controle manual de execução, como BFS, busca binária e redução. A sua ausência de overhead de runtime e controle explícito sobre a divisão de tarefas contribuem para sua eficiência em arquiteturas menores e médias. No entanto, esse modelo impõe um esforço maior de implementação e cuidados com sincronização, exigindo uma maior experiência do programador com controle manual de threads, o que pode comprometer sua escalabilidade em problemas mais dinâmicos.

Halide demonstrou ser a linguagem mais eficiente no contexto do processamento de imagens. Sua capacidade de expressar operações como filtros, transformações e binarização de maneira declarativa e otimizá-las automaticamente via parallel() e vectorize() proporcionou ganhos massivos de desempenho.

No entanto, sua eficácia não se estende da mesma forma para problemas com controle de fluxo dinâmico, como ordenações recursivas (Quicksort) e buscas em grafos, nos quais seu modelo de pipeline não se adapta bem. Isso quebra o paradigma de que todo programa que executa em paralelo terá ganhos significativos em relação a uma versão tradicional, sendo isso limitado a natureza da linguagem e o problema que está sendo explorado.

Dessa forma, é possível observar que não há uma única linguagem ou técnica que seja universalmente superior. O comportamento de cada abordagem depende fortemente do tipo de algoritmo, da estrutura do problema e da arquitetura do hardware. Julia, por exemplo, cresce em desempenho com o aumento de threads, sendo ideal para arquiteturas modernas e para algoritmos numéricos intensivos.

OpenMP se mantém como uma solução eficaz e prática para arquiteturas mais limitadas. Halide é extremamente eficaz em pipelines espaciais de imagem, mas pouco eficiente em algoritmos com ramificações e controle dinâmico de fluxo. C com threads oferece controle e desempenho, mas ao custo de maior complexidade de implementação.

Por fim, os resultados indicam que o desenvolvedor deve considerar cuidadosamente o perfil do problema e a arquitetura alvo ao escolher uma linguagem ou framework de paralelismo. A escolha equivocada pode neutralizar os benefícios da paralelização, enquanto uma combinação bem planejada pode proporcionar ganhos superiores a 1000 vezes, como demonstrado em várias técnicas de processamento de imagens.

Assim, a programação paralela se consolida não apenas como uma ferramenta poderosa, mas como uma necessidade estratégica para o desenvolvimento de aplicações eficientes e escaláveis no cenário computacional atual, principalmente com o avanço da inteligência artificial (IA) que demanda grandes volumes de dados como no mundo do entretenimento com tempos de respostas curtos a solicitação do usuário.

5. CONCLUSÃO

Este trabalho apresentou um estudo experimental sobre a aplicação da programação paralela em algoritmos de processamento de imagens e problemas computacionais de alto desempenho. Foram selecionados algoritmos representativos de ambas as categorias, como transformações radiométricas, filtragens espaciais, detecção de bordas, limiarização de Otsu, multiplicação de matrizes, algoritmos de ordenação, buscas, redução e simulação de Monte Carlo.

As versões paralelas, implementadas com Halide, OpenMP, Threads em C e Julia, foram comparadas com implementações sequenciais em Python e C, utilizando diferentes configurações de hardware. Os resultados mostraram que o uso de técnicas paralelas permite reduções expressivas no tempo de execução, com ganhos que chegam a ser superiores a mil vezes, a depender do algoritmo, linguagem e arquitetura utilizada. Halide destacou-se como a abordagem com melhor desempenho no processamento de imagens, enquanto Julia se mostrou altamente eficiente em operações envolvendo computação de alto desempenho, como multiplicação de matrizes. Como reforçado por Kirk e Hwu (2016), o paralelismo eficiente exige o balanceamento entre sobrecarga de sincronização, custo de desenvolvimento e escalabilidade.

Como trabalhos futuros, indicamos a experimentação dessas técnicas em dispositivos móveis, como smartphones, a fim de avaliar seu desempenho em ambientes com restrições de energia e recursos computacionais. Além disso, sugere-se a ampliação do estudo para arquiteturas heterogêneas envolvendo GPUs, explorando modelos híbridos de paralelismo e técnicas de programação como CUDA e OpenCL.

6. REFERÊNCIAS

Aprenda Go – “Entenda a diferença entre concorrência e paralelismo”. Link: [Entenda a diferença entre concorrência e paralelismo - Aprenda Golang](#)

Covap - Técnicas e conceitos de processamento de imagens e visão computacional. Link: [Material introdutório de Processamento Digital de Imagens e Visão Computacional](#)

Estimating the value of Pi using Monte Carlo. Link: [Estimating the value of Pi using Monte Carlo | GeeksforGeeks](#)

Fundamentos matemáticos de processamento de imagens - Limiar de Otsu. Link: [\(PDF\) Fundamentos matemáticos de processamento de imagens](#)

GILBERT, John R.; MOLER, Cleve B.; SCHREIBER, Robert. Sparse Matrices in MATLAB: Design and Implementation. *SIAM Journal on Matrix Analysis and Applications*, 1992.

GONZALEZ, Rafael C.; WOODS, Richard E. *Digital Image Processing*. 4. ed. Londres: Pearson, 2018. ISBN 978-1-292-22304-9.

GRAMA, Ananth; GUPTA, Anshul; KARYPIS, George; KUMAR, Vipin. *Introduction to Parallel Computing*. 2. ed. Addison-Wesley, 2003.

GRAMA, Ananth et al. *Introduction to Parallel Computing*. 2. ed. Addison-Wesley, 2003.

HENNESSY, John L.; PATTERSON, David A. *Computer Architecture: A Quantitative Approach*. 6. ed. Morgan Kaufmann, 2019.

KIRK, David B.; HWU, Wen-mei W. *Programming Massively Parallel Processors: A Hands-on Approach*. 3. ed. Morgan Kaufmann, 2016.

KNUTH, Donald E. *The Art of Computer Programming: Sorting and Searching*. v. 3, Addison-Wesley, 1998.

Limiar de Otsu - MATLAB. Link: [otsuthresh - Global histogram threshold using Otsu's method - MATLAB](#)

Mergesort and Recurrences. Link: [Merge sort - Wikipedia](#)

METROPOLIS, Nicholas et al. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 1953.

Multi-core Processor Market - Historic Data (2019-2024), Global Trends 2025, Growth Forecasts 2037. Link: [Multi-core Processor Market valued at \\$33.5 billion in 2024 | Eyes \\$146.18 billion by 2062](#)

Multiplicação de matrizes - operações e complexidade. Link: [Matrix multiplication - Wikipedia](#)

OpenMP Reduction Case Study. Link: [Microsoft PowerPoint - trapezoid.pptx](#)

Parallel Quick Sort Algorithm using OpenMP. Link: [IEEE Paper Template in A4 \(V1\)](#)

Petrobras will invest BRL 500 million in five supercomputers. Link: [Petrobras will invest BRL 500 million in five supercomputers](#)

Rob Pike. "Concorrência é lidar com várias coisas ao mesmo tempo; paralelismo é fazer várias coisas ao mesmo tempo". Link: [\(8\) Desvendando os Mistérios da Programação Concorrente e Paralela em C# - Códigos, Conceitos e Aplicações! | LinkedIn](#)

Taxonomia de Flynn. Link: [Taxonomia de Flynn – Wikipédia, a enciclopédia livre](#)

Time and Space Complexity Analysis of Quick Sort. Link: [Time and Space Complexity Analysis of Quick Sort | GeeksforGeeks](#)

Universidade Federal do Rio de Janeiro – MapReduce e Paralelização. Link: [Microsoft PowerPoint - metricas](#)

What's the cost of the IT skills gap? IDC says \$5.5 trillion by 2026. Link: [What's the cost of the IT skills gap? IDC says \\$5.5 trillion by 2026 | CIO Dive](#)

Linguagens

C. Link: [C \(programming language\) - Wikipedia](#)

Halide. Link: [Halide](#)

Julia. Link: [Multi-Threading · The Julia Language](#)

OpenMP. Link: [Home - OpenMP](#)

Pthreads. Link: [pthread_cond_init\(3\) - Linux manual page](#)

Python. Link: [3.13.3 Documentation](#)

7. APÊNDICES

Link de acesso aos dados coletados para as técnicas de Processamento de Imagens discutidas durante as análises. Link: [Dados Coletados - Google Drive](#)

Link de acesso ao GitHub onde estão armazenados os códigos desenvolvidos para as técnicas de Processamento de Imagens. Link:
<https://github.com/SebasNeto/Processamento-de-Imagens---Programacao-Paralela.git>

Link de acesso aos dados coletados para as técnicas de Computação de Alto Desempenho (HPC). Link:  [Dados-Coletados-Monografia](#)

Link de acesso ao GitHub onde estão armazenados os códigos desenvolvidos para as técnicas de Computação de Alto Desempenho (HPC). Link:
<https://github.com/SebasNeto/Programacao-Paralela-Aplicacoes.git>