

# **RELATÓRIO**

## **Trabalho Prático 2 de Sistemas Operacionais**

**André Kaled Duarte Coutinho Andrade,  
Miguel Oliveira Moraes de Souza,  
Pedro Henrique Belota Gadelha,  
Carlos Henrick Cavalcante Gomes**

<sup>1</sup>Instituto de Computação (IComp) – Universidade Federal do Amazonas (UFAM)  
Av. Gen. Rodrigo Octávio, 6200, Coroado I – 69080-900 – Manaus – AM – Brasil

`{andre.duarte, miguel.moraes, pedro.belota, carlos.gomes}@icomp.ufam.edu.br`

### **1. Introdução**

Este relatório apresenta o desenvolvimento e avaliação de três programas envolvendo processamento paralelo com threads. O objetivo é comparar desempenho entre versões sequenciais e paralelas, calcular a métrica de aceleração (speedup) e demonstrar uma aplicação prática onde o uso de threads traz vantagens claras em relação à abordagem sequencial. As atividades/questões envolvem:

1. Cálculo de produto escalar entre vetores.
2. Multiplicação de matrizes.
3. Quebra de uma senha PIN de 6 dígitos a partir de um hash.

### **2. Ambiente de testes**

#### **2.1. Computador A**

As especificações detalhadas do computador são:

- CPU: **12th Gen Intel(R) Core(TM) i7-12650H**
- Núcleos Físicos: 10
- Threads Lógicas: 16
- RAM: 8 GB
- Sistema Operacional: Arch Linux

#### **2.2. Computador B**

As especificações detalhadas do computador são:

- CPU: **Intel(R) Core(TM) i3-9100F CPU @ 3.60GHz**
- Núcleos Físicos: 4
- Threads Lógicas: 4
- RAM: 16 GB
- Sistema Operacional: Windows 10

Linguagens utilizadas: Para as questões 1 e 2 do trabalho foram usadas Java, já para a questão 3 foi usada a linguagem C.

### 3. Questão 1 – Produto Escalar

#### 3.1. Implementação Sequencial

A versão sequencial realiza o cálculo clássico do produto escalar percorrendo os vetores linearmente.

Código utilizado: ProdutoEscalarSeq.java

#### 3.2. Implementação Paralela

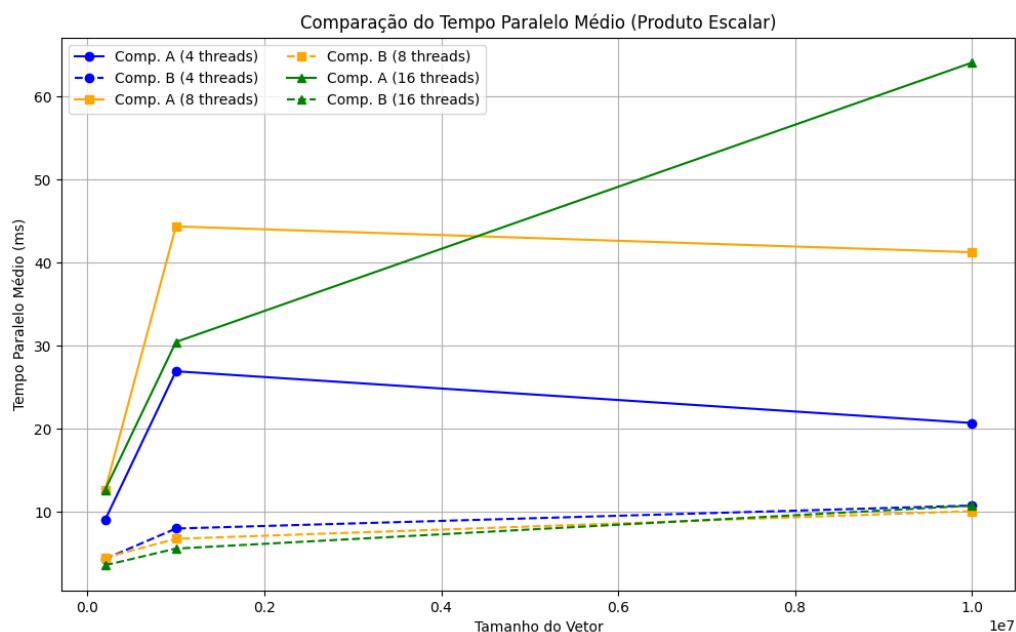
A versão paralela divide o vetor em blocos iguais entre as threads. Cada thread calcula uma soma parcial e o resultado final é obtido pela soma das parciais.

Código utilizado: ProdutoEscalarPar.java

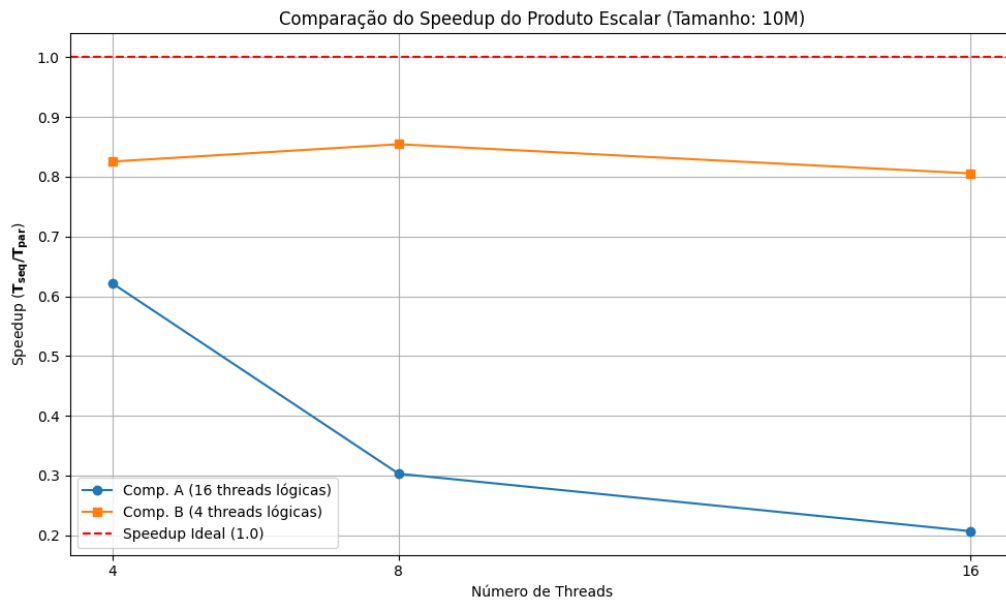
#### 3.3. Resultados coletados e Análise

Foram executados experimentos variando o tamanho dos vetores (200.000, 1.000.000 e 10.000.000) e a quantidade de threads (4, 8, 16).

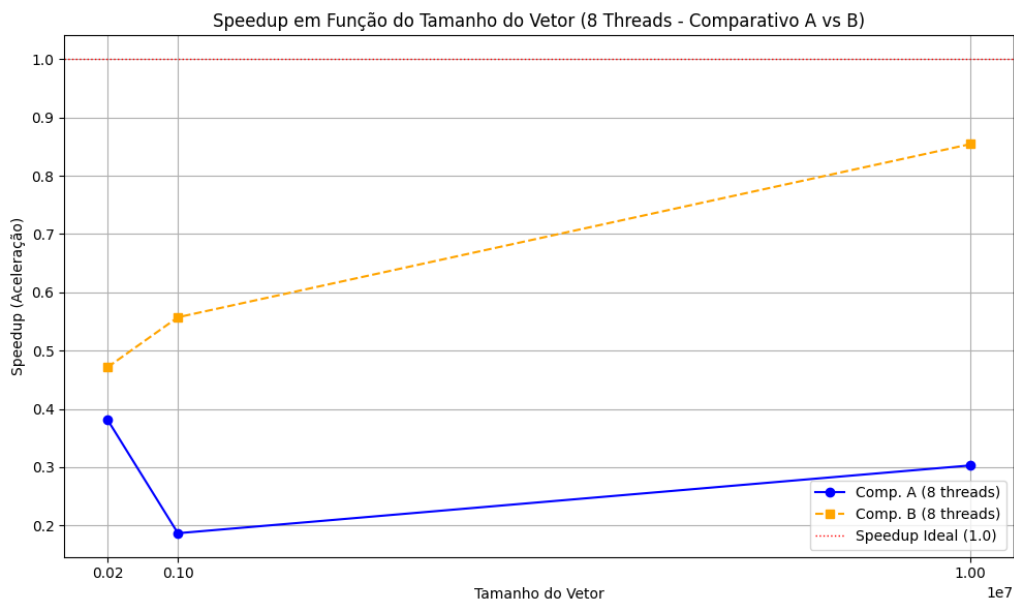
As Figuras 1, 2 e 3 apresentam os resultados gráficos.



**Figure 1. Comparação do Tempo Paralelo Médio (Produto Escalar) em função do Tamanho do Vetor.**



**Figure 2. Speedup obtido para o Produto Escalar (Tamanho: 10M) em função do número de threads.**



**Figure 3. Speedup em função do Tamanho do Vetor (8 Threads - Comparativo A vs B).**

### 3.3.1. Análise

A análise do speedup e do tempo absoluto (Figuras 1, 2, 3) deixa claro que o paralelismo não trouxe ganhos para o produto escalar nas condições testadas. Em todos os cenários, o tempo paralelo ficou acima do tempo sequencial, resultando em speedups menores que 1. Isso ocorre porque o custo de criação, gerenciamento e sincronização das threads (overhead) supera o trabalho efetivo feito por cada uma. O produto escalar é uma operação

simples por elemento, com baixa carga computacional ( $O(N)$ ) e forte dependência de memória, o que limita qualquer vantagem do paralelismo. O aumento no número de threads (especialmente no Computador A, Figura 2) piorou ainda mais o desempenho por aumentar o overhead e gerar contenção na soma das parciais. Em resumo, o problema não apresenta granularidade suficiente para justificar paralelização.

**Observação:** Embora o Computador B (com apenas 4 threads lógicas) seja significativamente mais rápido em tempo absoluto (Figura 1), a curva de Speedup de ambos permanece abaixo de 1.0, indicando a ineficiência inerente à granularidade do problema.

## 4. Questão 2 – Multiplicação de Matrizes

### 4.1. Implementação Sequencial

A versão sequencial calcula a multiplicação clássica  $C = A \times B$  percorrendo linha por linha de  $A$  e coluna por coluna de  $B$ . Cada elemento  $C_{ij}$  é obtido pela soma dos produtos entre a linha  $i$  de  $A$  e a coluna  $j$  de  $B$ .

Código utilizado: ProdutoMatrizSeq.java

### 4.2. Implementação Paralela

A versão paralela utiliza uma abordagem inspirada no modelo de execução em blocos do CUDA [NVIDIA Corporation]. A matriz resultado é dividida em blocos quadrados de tamanho fixo, e cada bloco é calculado por uma tarefa independente executada dentro de um *pool* fixo de *threads*. Essa técnica reduz acessos redundantes e melhora a distribuição de trabalho entre as threads, simulando a estratégia de grids e blocos de GPUs, mas adaptada para CPU.

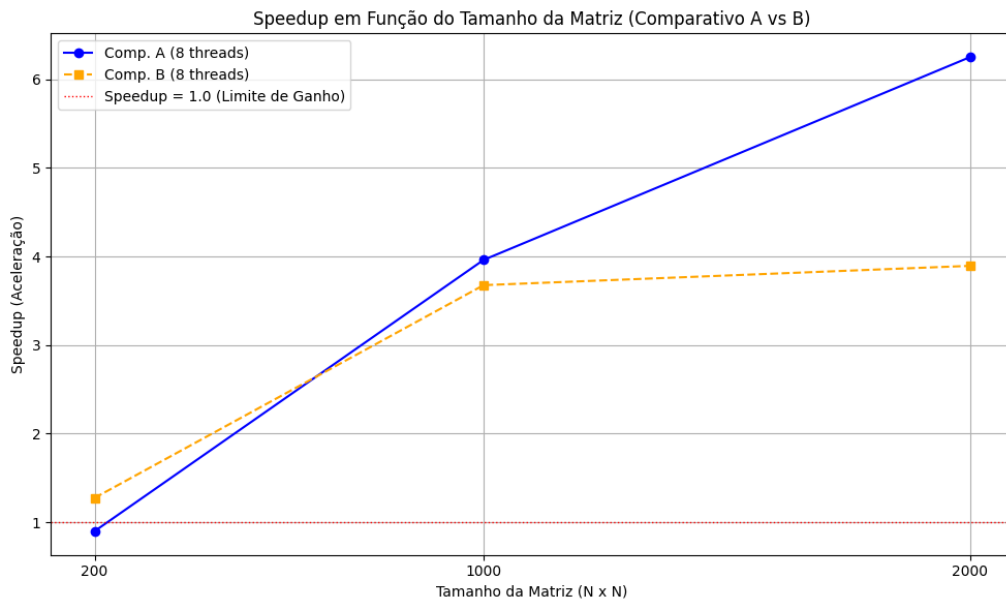
A abordagem de blocos foi inspirada em trabalhos sobre multiplicação de matrizes em CUDA [Kharshit 2024] e na documentação oficial da NVIDIA [NVIDIA Corporation]. Exemplos práticos podem ser encontrados nos repositórios de código da NVIDIA [NVIDIA].

Código utilizado: ProdutoMatrizPar.java

### 4.3. Resultados coletados e Análise

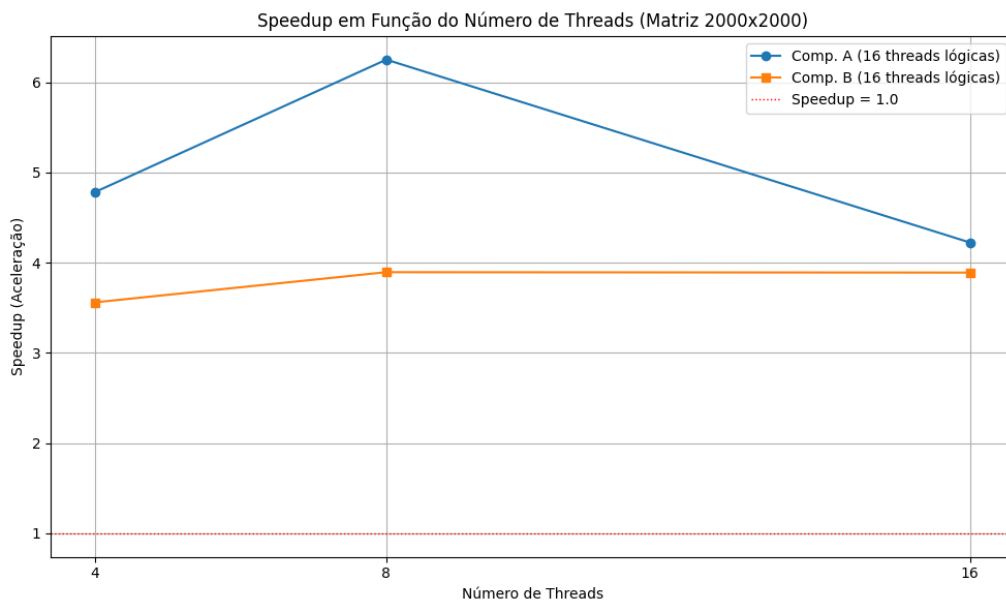
Foram executados experimentos variando o tamanho da matriz (200 X 200, 1.000 X 1.000 e 2.000 X 2.000) e a quantidade de threads (4, 8, 16).

As Figuras 4, 5 e 6 apresentam os resultados gráficos.



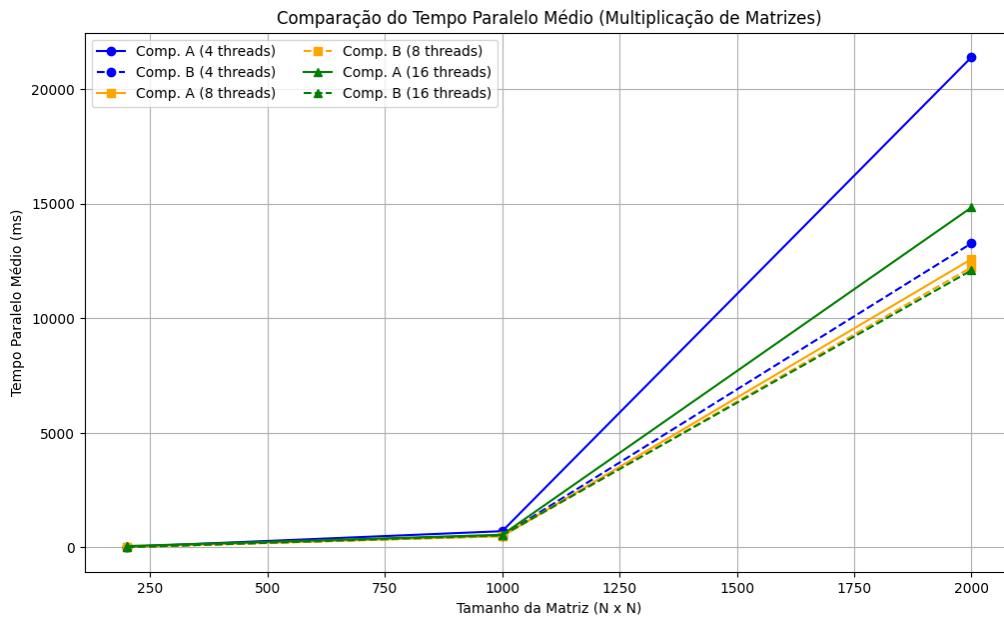
**Figure 4. Speedup em Função do Tamanho da Matriz (Comparativo A vs B, 8 threads).**

O gráfico mostra que o speedup aumenta drasticamente com o tamanho da matriz, demonstrando que o problema possui granularidade ideal para paralelismo.



**Figure 5. Speedup em Função do Número de Threads (Matriz 2000x2000).**

O speedup obtido para matrizes grandes é significativamente maior que 1, reforçando que o paralelismo compensou o custo adicional e resultou em um grande ganho de desempenho.



**Figure 6. Comparação do Tempo Paralelo Médio (Multiplicação de Matrizes) em função do Tamanho.**

#### 4.3.1. Análise da Escalabilidade

O problema de multiplicação de matrizes ( $O(N^3)$ ) apresenta uma granularidade de trabalho elevada e uma complexidade computacional que justifica o uso de paralelismo. Diferente do produto escalar, os resultados mostram um ganho de desempenho significativo, especialmente para matrizes grandes (Figura 4). **Pico de Speedup (Computador A):** O pico de desempenho foi alcançado no Computador A com 8 threads, resultando em um Speedup de  $6.25\times$  (Figura 5), demonstrando que, para um problema computacionalmente caro, o tempo de processamento supera o overhead. O tempo sequencial ( $T_{seq}$ ) foi reduzido de  $\approx 78$  segundos para  $\approx 12.5$  segundos.

**Queda do Desempenho (8 para 16 threads):** A partir de 8 threads no Comp. A, o speedup decai de  $6.25\times$  para  $4.22\times$ . Esta queda é um sinal de saturação de recursos e contenção de cache. Com 16 threads ativas, a sobrecarga de gerenciamento (troca de contexto) e a competição pelo cache da CPU se tornam mais custosas do que o tempo economizado com a divisão do trabalho, limitando o ganho de paralelismo e levando ao estado de ineficiência.

**Comparação Absoluta (Comp. A vs Comp. B):** Apesar do Speedup relativo do Comp. A ser maior, o Tempo Paralelo Médio absoluto do Computador B é ligeiramente melhor na matriz  $2000\times 2000$  (Figura 6). Isso se deve à combinação do maior desempenho de núcleo único e da maior capacidade de RAM do Computador B, que otimiza o tempo sequencial de base ( $T_s$ ), mesmo com apenas 4 threads lógicas.

### 5. Questão 3 – Força bruta de PIN (sequencial $\times$ paralelo)

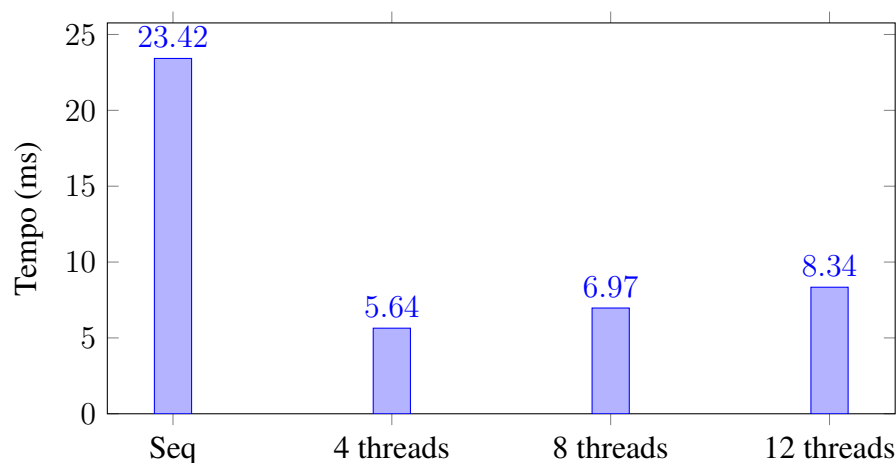
Nesta questão foi implementado um ataque de força bruta para quebrar um PIN numérico de 6 dígitos (000000–999999). O programa primeiro recebe um PIN do usuário, calcula

o hash com a função `hash_senha` e grava o valor em `hash.txt`. Depois, oferece duas formas de tentar descobrir a senha original: uma versão sequencial e uma versão paralela usando `pthreads`.

Na versão sequencial o programa percorre o espaço de busca de 000000 a 999999. Para cada valor  $i$  ele gera a string do PIN, calcula o hash e compara com o hash alvo. O laço termina quando encontra a senha ou quando esgota todas as combinações. O tempo é medido com `clock_gettime` e convertido para milissegundos.

Na versão paralela esse mesmo espaço de busca é dividido entre  $N$  threads. Cada thread recebe um intervalo `[início, fim]` e roda o laço apenas naquele intervalo. As threads compartilham uma flag encontrou e um buffer com a senha encontrada. A primeira versão usava `pthread_mutex` para proteger essa flag a cada iteração, além de vários `printf` dentro do laço. O custo de sincronização e de I/O era tão alto que o código paralelo acabava ficando mais lento que o sequencial. Na versão final isso foi trocado por operações atômicas (`atomic_load` e `atomic_exchange`) e os `printf` do laço foram removidos, deixando o paralelismo bem mais leve.

## Resultados



**Figure 7. Comparação entre a versão sequencial (senha intermediária) e a versão paralela (564321).**

Para a versão sequencial foram testadas três senhas em posições diferentes da faixa: uma mais próxima do início, uma intermediária e uma mais ao final. Os tempos estão resumidos na Tabela 1.

**Table 1. Tempos da versão sequencial para diferentes senhas (PIN de 6 dígitos).**

Senha	Posição aproximada	Tempo (ms)
123412	Início	9,586
567855	Intermediária	23,420
987965	Final	37,958

A Tabela 2 mostra os resultados da versão paralela usando uma senha intermediária (564321) e variando apenas o número de *threads*. Como referência, o tempo sequencial da senha intermediária foi de aproximadamente 23,4 ms.

**Table 2. Tempos da versão paralela para a senha 564321, variando o número de *threads*.**

# Threads	Tempo (ms)	Aceleração vs. sequencial
4	5,641	$\approx 4,15\times$
8	6,970	$\approx 3,36\times$
12	8,341	$\approx 2,81\times$

Com as otimizações (remoção dos `mutex` no laço e uso de variáveis atômicas), a versão paralela finalmente passou a compensar: com 4 *threads*, o tempo caiu de cerca de 23,4 ms para 5,6 ms. A partir daí o ganho começa a diminuir, porque o número de *threads* fica grande em relação ao hardware e o custo de gerenciar as *threads* começa a aparecer. Mesmo assim, os resultados mostram que, para um espaço de busca de 6 dígitos, o paralelismo bem usado consegue reduzir de forma significativa o tempo de quebra por força bruta.

### References

Kharshit (2024). Matrix multiplication cuda. <https://kharshit.github.io/blog/2024/06/07/matrix-multiplication-cuda>. Acesso em: 29/11/2025.

NVIDIA. Cuda samples. <https://github.com/NVIDIA/cuda-samples>. Acesso em: 29/11/2025.

NVIDIA Corporation. Cuda c programming guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acesso em: 29/11/2025.