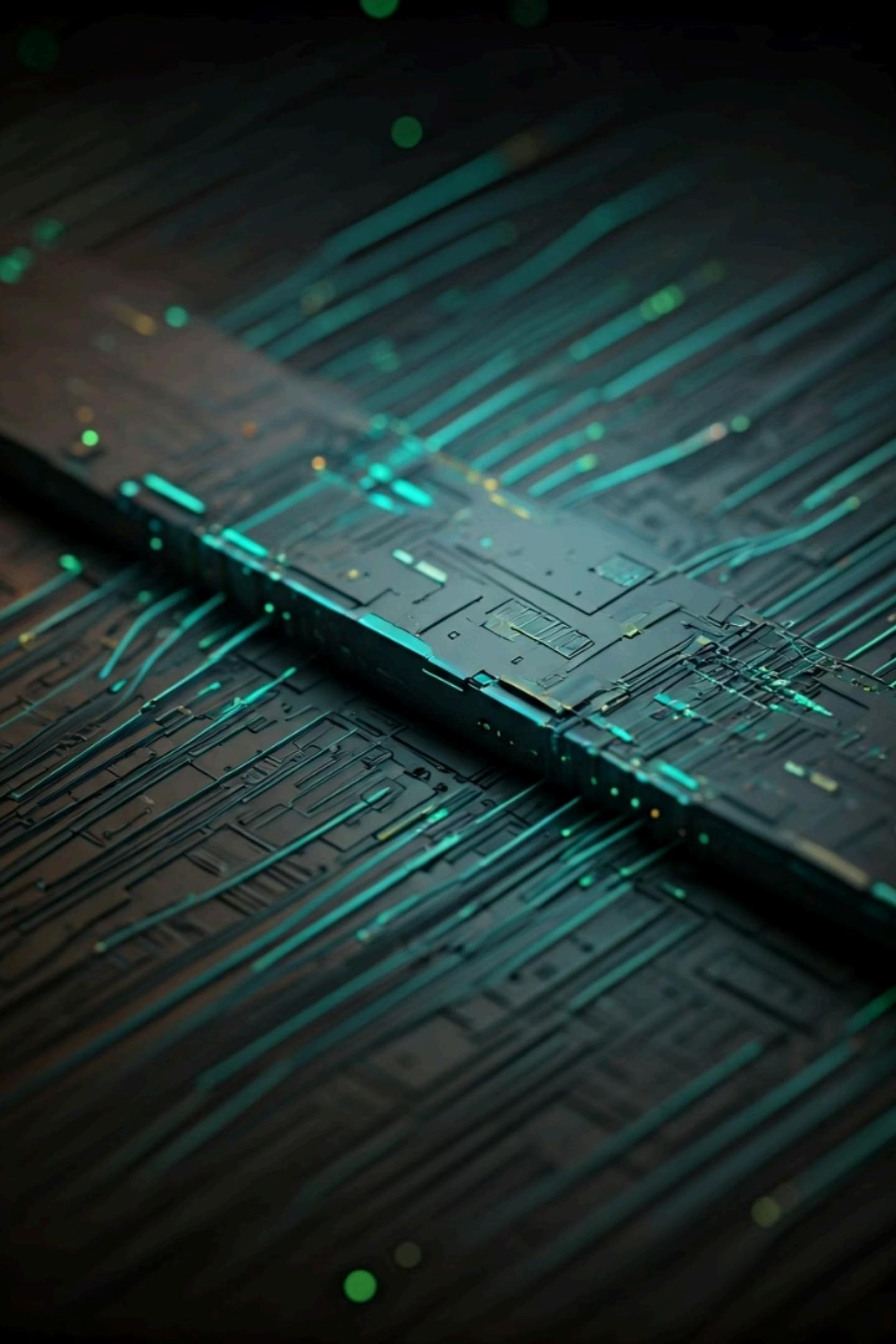


Multiplicação de Matrizes com Cálculo Paralelo usando Threads

Compontes: Guilherme Ramos e Lucas Nobre



O que é Multiplicação de Matrizes?

Definição Fundamental

Multiplicar matrizes consiste em combinar linhas de A com colunas de B para formar uma nova matriz C.

É uma operação comum, mas custosa para matrizes grandes.

Operação entre duas matrizes A ($m \times n$) e B ($n \times p$).

Gera uma matriz C ($m \times p$).

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$



Custo Computacional



Complexidade $O(n^3)$

Multiplicação tradicional.

Tempo Cresce Rapidamente

Aumento exponencial com o tamanho da matriz.

Solução: Paralelização

Distribuir o trabalho.

A multiplicação tem complexidade $O(n^3)$. Isso significa que, conforme as dimensões aumentam, o tempo de execução cresce exponencialmente.

Paralelismo com Threads

Distribuição de Trabalho

Múltiplas threads envolvidas.



Cada elemento da matriz resultado $C[i][j]$ é calculado como o somatório do produto dos elementos da linha i de A com a coluna j de B .

Ou seja: linha de $A \times$ coluna de $B =$ elemento de C

Atribuição de Tarefas

Cada thread calcula linhas ou blocos.

Redução de Tempo

Execução significativamente mais rápida.

Paralelismo com Threads

Cada thread recebe um conjunto de linhas da matriz resultado C para calcular.

Por exemplo, com 8 threads e 800 linhas, cada uma calcula 100 linhas.

Dentro dessas linhas, cada elemento $C[i][j]$ é obtido multiplicando a linha i de A pela coluna j de B^T .

Essa divisão garante que nenhuma thread interfere no trabalho da outra, evitando conflitos e mantendo o desempenho.

Arquitetura do Projeto



`utils.c / utils.h`

Funções auxiliares.

`sequencial.c`

Implementação sem threads.

`paralelo.c`

Uso de `pthread.h`.

`benchmark.py`

Script de comparação.

O projeto é modular: código sequencial e paralelo separados, funções auxiliares em `utils`, e um script Python para automação dos testes.

Os códigos

```
1 #ifndef UTILS_H
2 #define UTILS_H
3
4 int** alocar_matriz(int linhas, int colunas);
5 void liberar_matriz(int** matriz, int linhas);
6 void preencher_matriz(int** matriz, int linhas, int colunas);
7 void salvar_matriz(const char* nome_arquivo, int** matriz, int linhas, int colunas);
8 int** carregar_matriz(const char* nome_arquivo, int linhas, int colunas);
9
10#endif
11
12int** alocar_matriz(int linhas, int colunas) {
13    int** matriz = (int**) malloc(linhas * sizeof(int*));
14    for (int i = 0; i < linhas; i++) {
15        matriz[i] = (int*) malloc(colunas * sizeof(int));
16    }
17    return matriz;
18}
19
20void liberar_matriz(int** matriz, int linhas) {
21    for (int i = 0; i < linhas; i++) {
22        free(matriz[i]);
23    }
24    free(matriz);
25}
26
27void preencher_matriz(int** matriz, int linhas, int colunas) {
28    for (int i = 0; i < linhas; i++) {
29        for (int j = 0; j < colunas; j++) {
30            matriz[i][j] = rand() % 10;
31        }
32    }
33}
34
35void salvar_matriz(const char* nome_arquivo, int** matriz, int linhas, int colunas) {
36    FILE* f = fopen(nome_arquivo, "w");
37    for (int i = 0; i < linhas; i++) {
38        for (int j = 0; j < colunas; j++) {
39            fprintf(f, "%d ", matriz[i][j]);
40        }
41        fprintf(f, "\n");
42    }
43    fclose(f);
44}
45
46int** carregar_matriz(const char* nome_arquivo, int linhas, int colunas) {
47    FILE* f = fopen(nome_arquivo, "r");
48    int** matriz = alocar_matriz(linhas, colunas);
49    for (int i = 0; i < linhas; i++) {
50        for (int j = 0; j < colunas; j++) {
51            fscanf(f, "%d", &matriz[i][j]);
52        }
53    }
54    fclose(f);
55    return matriz;
56}
```

```

def run_program(cmd):
    result = subprocess.run(cmd, capture_output=True, text=True)
    return result.stdout.strip()

def extrair_tempo(texto):
    match = re.search(r'Tempo .*: ([0-9.]+) segundos', texto)
    if match:
        return float(match.group(1))
    else:
        return None

def main():
    tamanho = 3000

    print(f"\n==== Benchmark Multiplicacao de Matrizes ({tamanho}x{tamanho}) ===\n")

    print("Versao Sequencial:")
    out_seq = run_program(["./sequencial", str(tamanho)])
    print(out_seq)
    tempo_seq = extrair_tempo(out_seq)

    print("\nVersao Paralela (8 threads):")
    out_par = run_program(["./paralelo", str(tamanho)])
    print(out_par)
    tempo_par = extrair_tempo(out_par)

    metodos = ['Sequencial', 'Paralelo (8 threads)']
    tempos = [tempo_seq, tempo_par]

    plt.figure(figsize=(8,5))
    plt.bar(metodos, tempos, color=['blue', 'green'])
    plt.ylabel('Tempo (segundos)')
    plt.title(f'Benchmark Multiplicação de Matrizes {tamanho}x{tamanho}')
    for i, v in enumerate(tempos):
        plt.text(i, v + max(tempos)*0.01, f"{v:.2f}s", ha='center', fontweight='bold')
    plt.show()

if __name__ == "__main__":
    main()

```

Benchmark em Python



Arquivo benchmark.py

Script principal.



Execução subprocess

Chama os programas C.



Medição de tempo

gettimeofday()

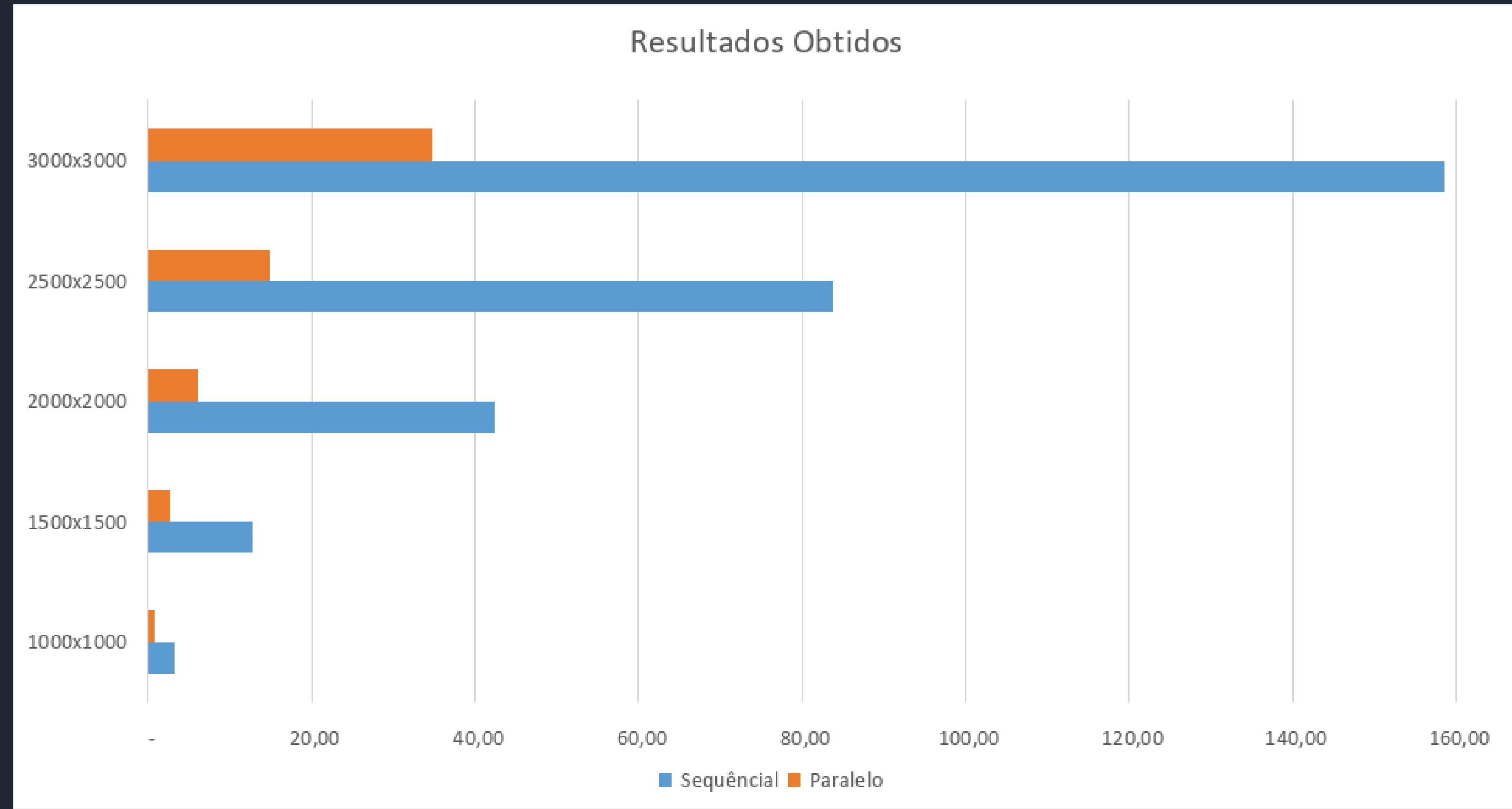


Exemplo de uso

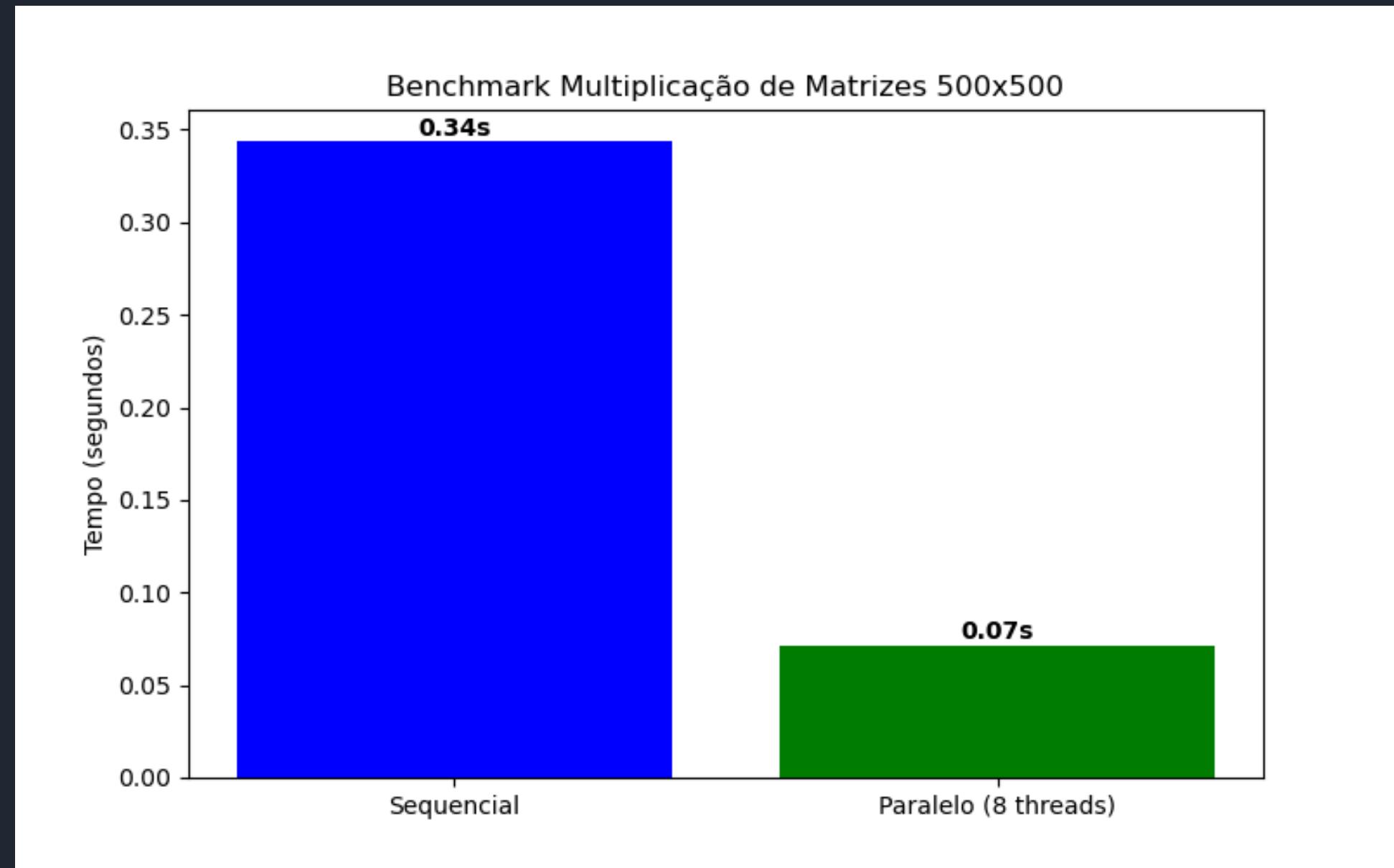
python benchmark.py

benchmark.py executa ambos os binários com os mesmos dados e mede o tempo de forma precisa. Os dados são usados para análise e gráficos.

Resultados Obtidos

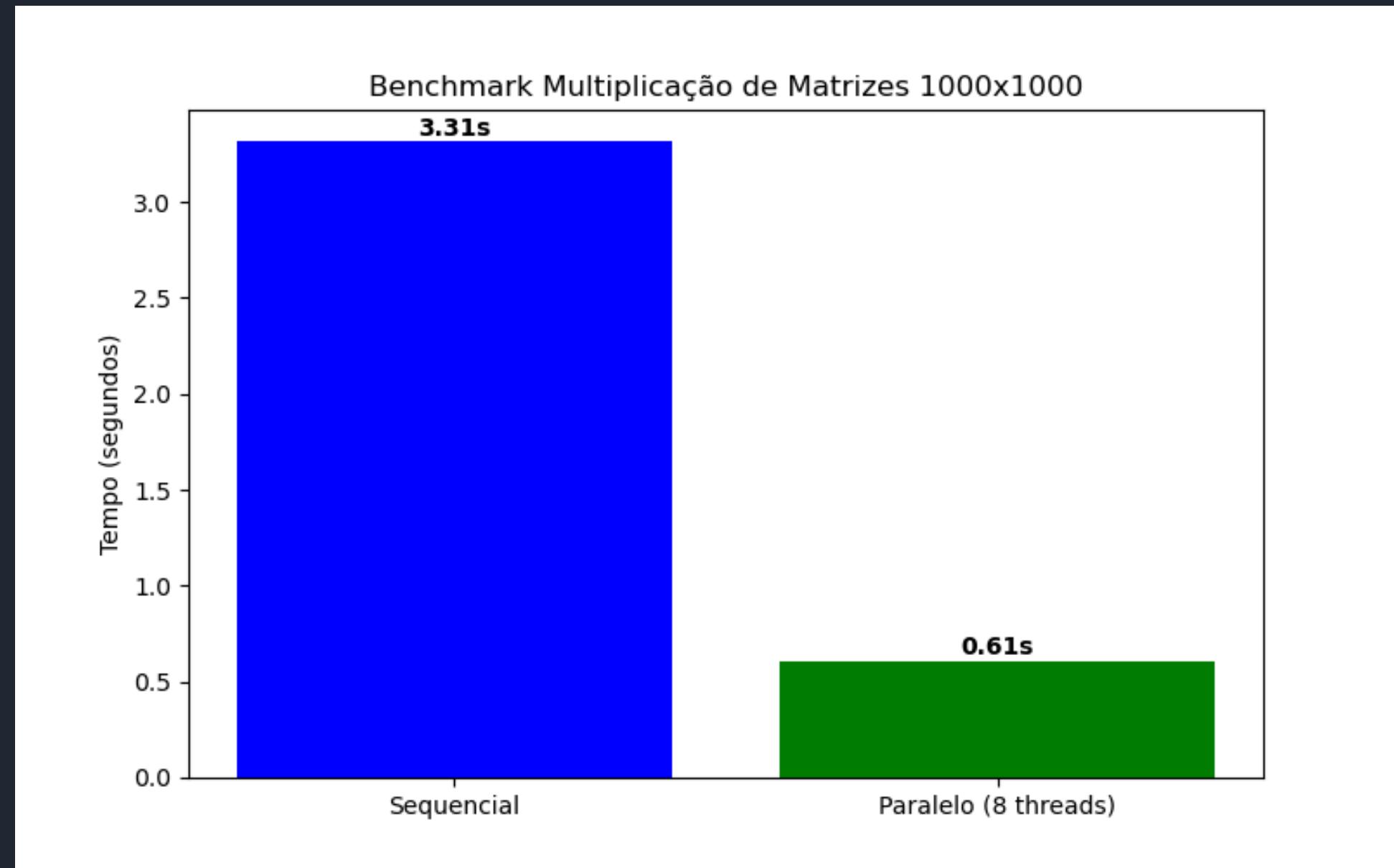


Resultados Obtidos



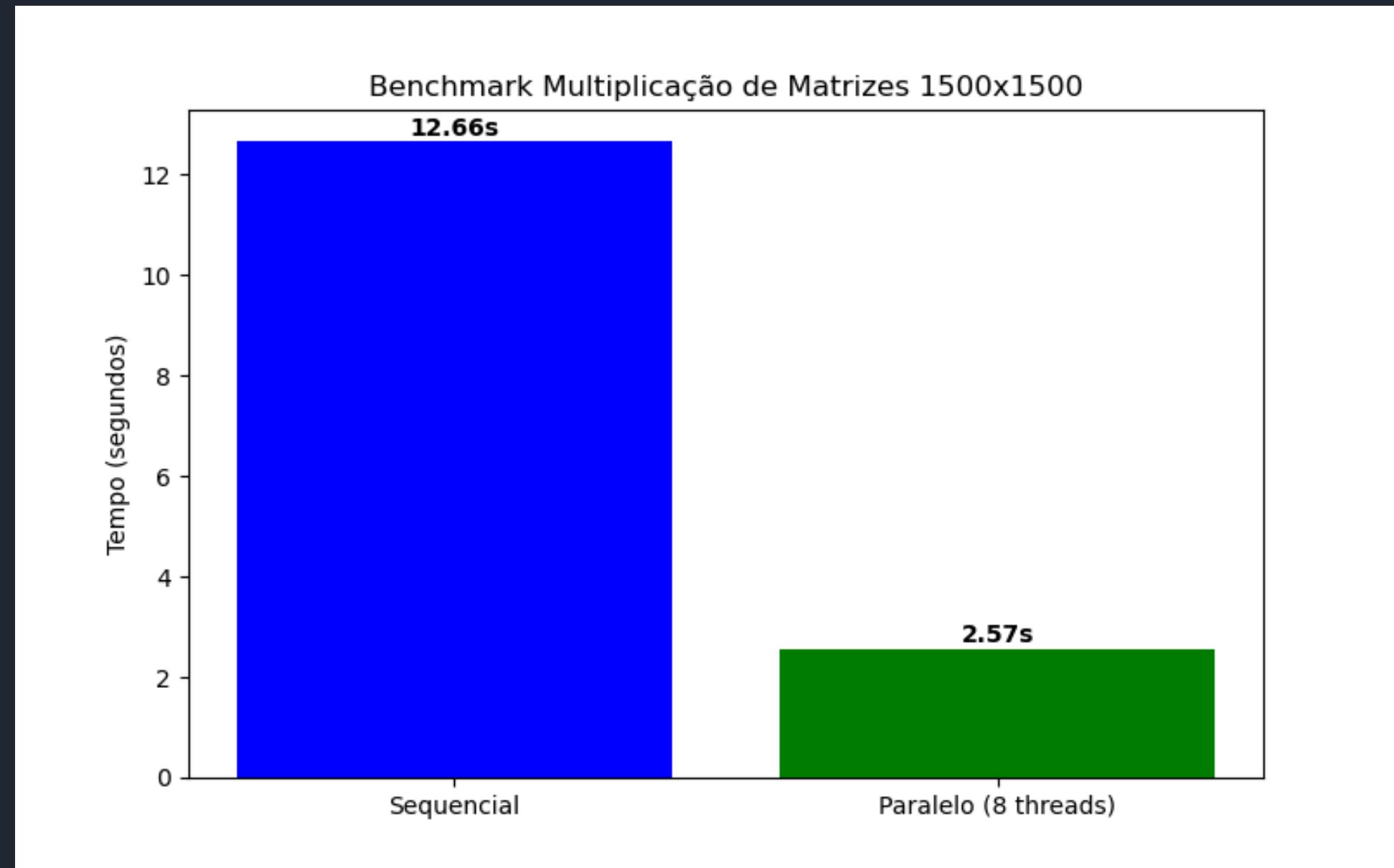
A abordagem paralela mostra melhorias claras no desempenho. Quanto maior a matriz, maior o ganho em tempo usando múltiplas threads.

Resultados Obtidos



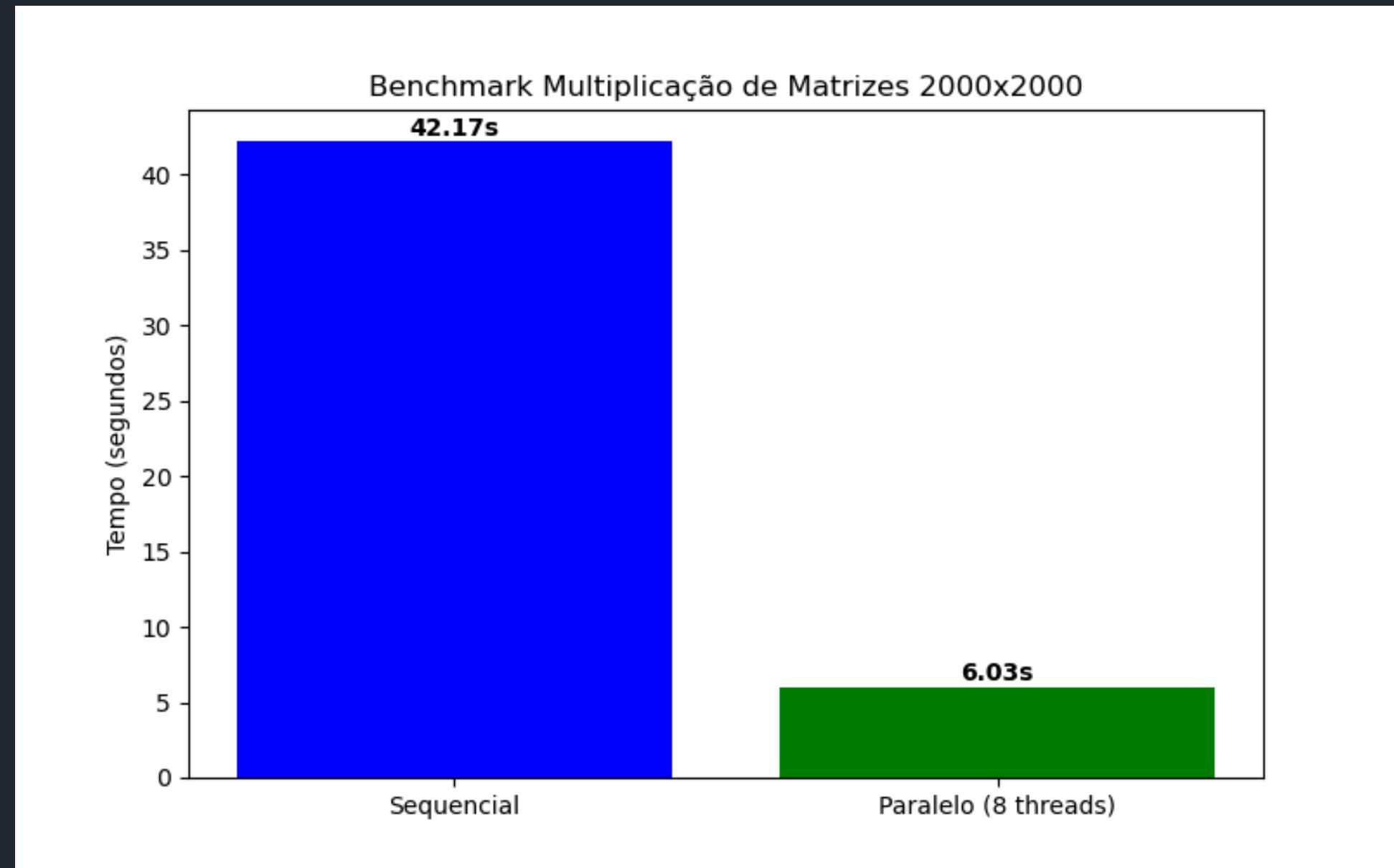
A abordagem paralela mostra melhorias claras no desempenho. Quanto maior a matriz, maior o ganho em tempo usando múltiplas threads.

Resultados Obtidos



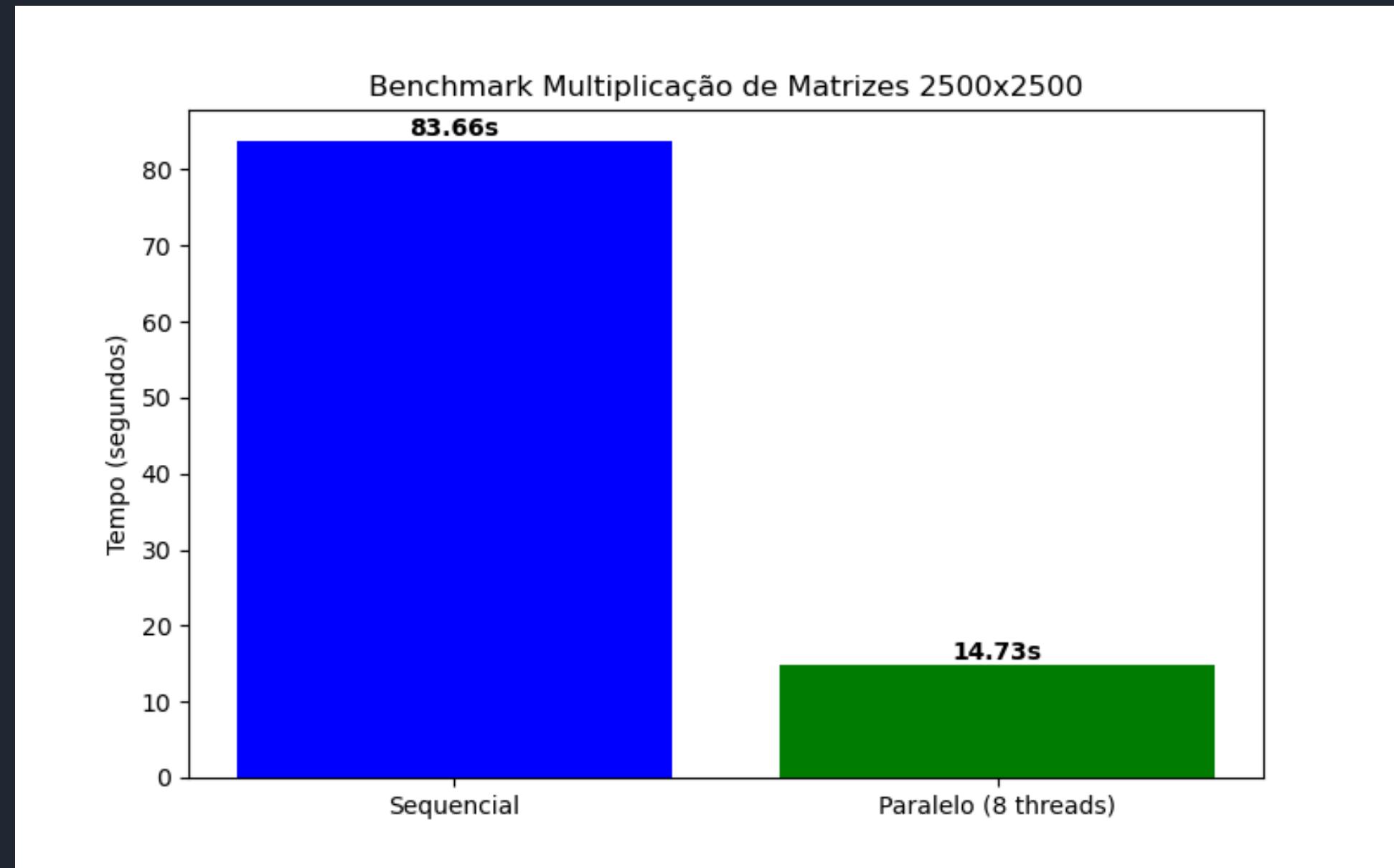
A abordagem paralela mostra melhorias claras no desempenho. Quanto maior a matriz, maior o ganho em tempo usando múltiplas threads.

Resultados Obtidos



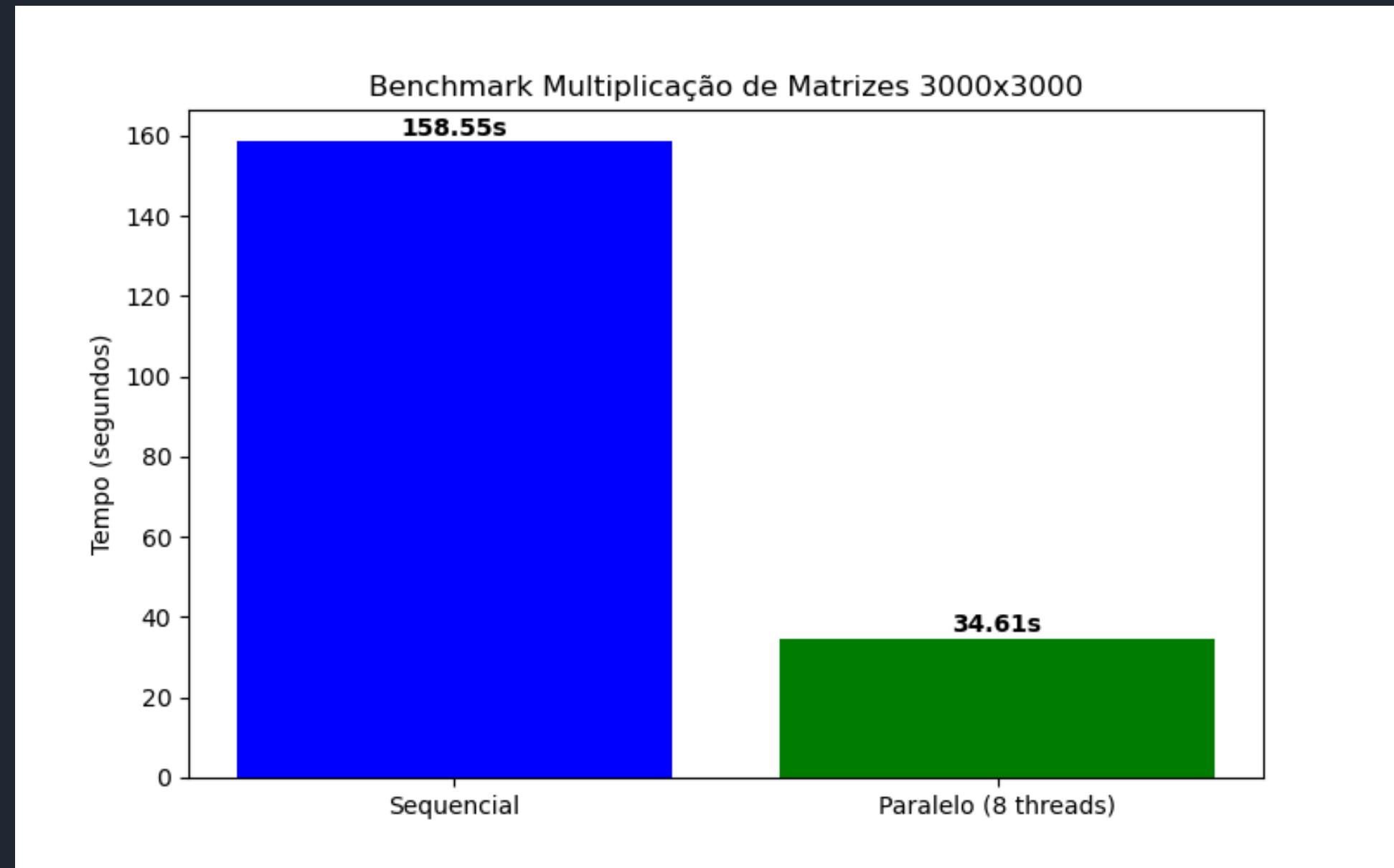
A abordagem paralela mostra melhorias claras no desempenho. Quanto maior a matriz, maior o ganho em tempo usando múltiplas threads.

Resultados Obtidos



A abordagem paralela mostra melhorias claras no desempenho. Quanto maior a matriz, maior o ganho em tempo usando múltiplas threads.

Resultados Obtidos



A abordagem paralela mostra melhorias claras no desempenho. Quanto maior a matriz, maior o ganho em tempo usando múltiplas threads.

Considerações Técnicas



Sem Deadlock

Acesso a linhas distintas.



Memória Controlada

alocação e liberação das matrizes

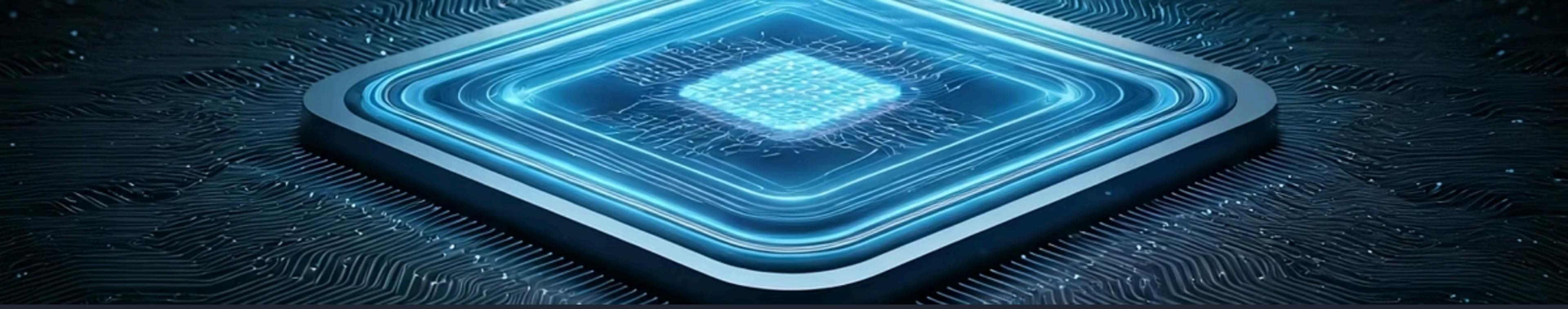


Cuidado com Grandes Matrizes

Alto consumo de memória RAM e tempo.

O paralelismo é seguro pois cada thread calcula regiões exclusivas. Atenção à RAM e ao número de threads para não sobrecarregar o sistema.





Conclusões

Paralelização é Essencial

Redução drástica no tempo de execução.

Ideal para Matrizes Grandes
Acima de 1000x1000.

O uso de threads é altamente vantajoso em matrizes grandes. A combinação de C para performance e Python para análise torna o projeto eficiente e didático.

Python como Facilitador
Benchmark e visualização.