Análise de Desempenho

Antonio Neto

2024

Resumo

Este relatório tem como objetivo explorar os conceitos de análise de desempenho de algoritmos, utilizando ferramentas como *time* e *perf* em um ambiente Linux, especificamente na distribuição Ubuntu 20.04 do Docker (comandos para execução no README.md). O estudo foca na implementação de algoritmos para cálculo do fatorial de um número, utilizando métodos recursivo e iterativo, bem como na comparação de desempenho entre eles. Adicionalmente, foram analisados algoritmos de divisão e operações de escrita em arquivo. **O DockerRun.txt tem os dados brutos da execução do container**.

Sumário

1 Desenvolvimento

Os experimentos foram realizados em um container Docker. Os códigos em C foram desenvolvidos para calcular o fatorial de 10,20,30,40,50,60,70,80,90 e 100 de forma recursiva e iterativa, realizar operações de divisão de inteiros e floats, e operações de escrita em arquivo. Scripts em Python foram utilizados para automatizar a execução dos testes, coletar e analisar os dados, e gerar gráficos comparativos. Vale mencionar que os scripts para calcular o fatorial não são convencionais, pois utilizei vetores para armazenar os dígitos do resultado, permitindo calcular fatoriais de números que usariam bem mais de 64 bits. Também é preciso considerar que os códigos estão sendo executados em ambiente virtualizado, e não refletem a realidade de modo absoluto!

1.1 Implementação dos Algoritmos

Os algoritmos foram implementados conforme descrito abaixo:

1.1.1 Algoritmo Recursivo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_DIGITS 5000

void multiply(int result[], int *result_size, int x) {
    int carry = 0;
    for (int i = 0; i < *result_size; ++i) {
        int prod = result[i] * x + carry;
        result[i] = prod % 10;
        carry = prod / 10;
    }

    while (carry) {
        result[(*result_size)++] = carry % 10;
        carry /= 10;
    }
}

void factorial_recursive(int result[], int *result_size, int n) {
    if (n == 1) {
        return;
    }

    factorial_recursive(result, result_size, n - 1);
    multiply(result, result_size, n);
}

void print_result(int result[], int result_size) {
    for (int i = result_size - 1; i >= 0; --i) {
```

```
        printf("%d", result[i]);
    }
    printf("\n");
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }

    int num = atoi(argv[1]);
    if (num < 0) {
        printf("Please enter a non-negative integer.\n");
        return 1;
    }

    int result[MAX_DIGITS];
    memset(result, 0, sizeof(result));
    result[0] = 1;
    int result_size = 1;

    factorial_recursive(result, &result_size, num);

    print_result(result, result_size);

    return 0;
}
```

1.1.2 Algoritmo Iterativo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_DIGITS 5000

void multiply(int result[], int *result_size, int x) {
    int carry = 0;
    for (int i = 0; i < *result_size; ++i) {
        int prod = result[i] * x + carry;
        result[i] = prod % 10;
        carry = prod / 10;
    }

    while (carry) {
        result[(*result_size)++] = carry % 10;
        carry /= 10;
```

```
    }
}

void factorial_iterative(int result[], int *result_size, int n) {
    for (int x = 2; x <= n; ++x) {
        multiply(result, result_size, x);
    }
}

void print_result(int result[], int result_size) {
    for (int i = result_size - 1; i >= 0; --i) {
        printf("%d", result[i]);
    }
    printf("\n");
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }

    int num = atoi(argv[1]);
    if (num < 0) {
        printf("Please enter a non-negative integer.\n");
        return 1;
    }

    int result[MAX_DIGITS];
    memset(result, 0, sizeof(result));
    result[0] = 1;
    int result_size = 1;

    factorial_iterative(result, &result_size, num);

    print_result(result, result_size);

    return 0;
}
```

Grafico representando o comando time executado nos códigos acima de fatorial. Demonstrou ser pouco preciso.

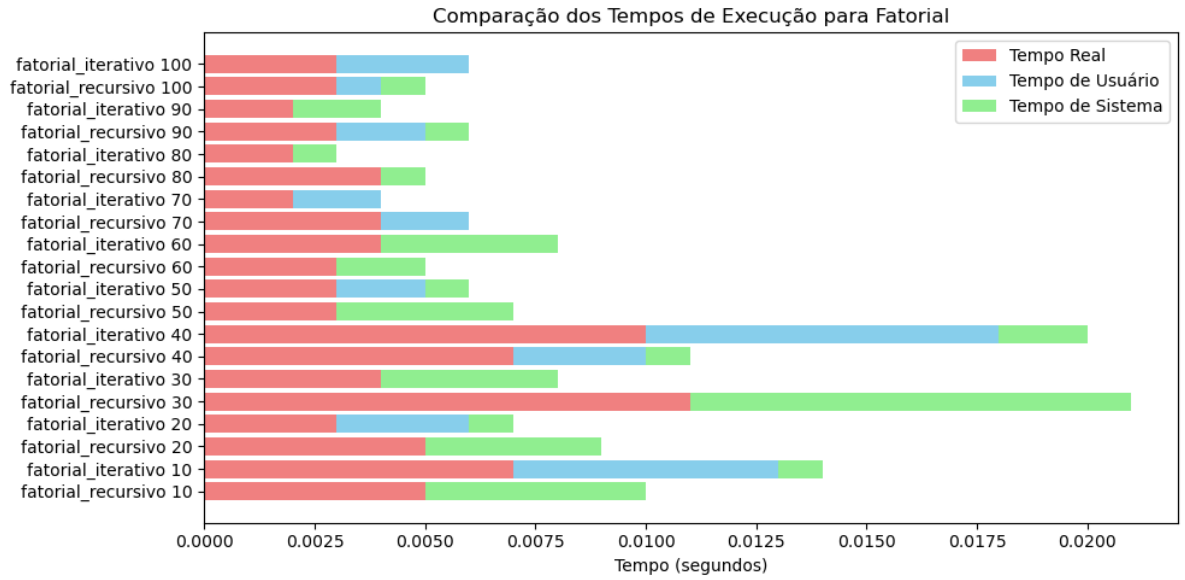

Figura 1: Tempo de Execução dos Algoritmos de Fatorial (time)

1.1.3 Divisão de Inteiros

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 1000000; ++i) {
        int a = i, b = i + 1;
        int c = a / b;
        int d = b / a;
        int e = a / a;
        int f = b / b;
    }
    return 0;
}
```

1.1.4 Divisão de Floats

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 1000000; ++i) {
        float a = (float)i, b = (float)(i + 1);
        float c = a / b;
        float d = b / a;
        float e = a / a;
        float f = b / b;
    }
```

```
    return 0;
}
```

1.1.5 Operações de Escrita em Arquivo

```
#include <stdio.h>

int main() {
    FILE *file = fopen("output.txt", "w");
    if (!file) {
        perror("Erro ao abrir o arquivo");
        return 1;
    }

    for (int i = 1; i <= 1000000; ++i) {
        int a = i, b = i + 1;
        int c = a / b;
        fprintf(file, "%d\n", c);
    }

    fclose(file);
    return 0;
}
```

1.2 Execução dos Testes

Os testes foram realizados para entradas variando de 10 a 100 no caso dos algoritmos de fatorial, e para um loop de 1 a 1 milhão nos casos das divisões e operações de escrita. As ferramentas *time* e *perf* foram utilizadas para medir o tempo de execução e coletar métricas de desempenho. O script *run_tests.sh* foi utilizado para automatizar a execução dos testes e salvar os resultados.

1.3 Coleta e Análise dos Dados

Os resultados dos testes foram organizados no arquivo **DockerRun.json**, que foi posteriormente analisado utilizando o plot.py. Este script utilizou a biblioteca *matplotlib* para gerar gráficos comparativos dos tempos de execução e outras métricas de desempenho:

```
import json
import matplotlib.pyplot as plt
import numpy as np
import os
import re

if not os.path.exists('resultados'):
    os.makedirs('resultados')

with open('DockerRun.json') as f:
    data = json.load(f)
```

```
executions = data['executions']
metrics = ['task_clock', 'cycles', 'instructions', 'branches', 'branch_misses', 'l1_d

metrics_data = {metric: [] for metric in metrics}
names = []
real_times = []
user_times = []
sys_times = []

def parse_time(time_str):
    # Extract only the numeric part and convert it to float
    return float(re.findall(r"[-+]?\d*\.\d+|\d+", time_str)[0])

for execution in executions:
    names.append(f"{execution['name']} {execution.get('input', '')}".strip())
    real_time_str = execution['real_time']
    user_time_str = execution['user_time']
    sys_time_str = execution['sys_time']
    real_times.append(parse_time(real_time_str))
    user_times.append(parse_time(user_time_str))
    sys_times.append(parse_time(sys_time_str))
    for metric in metrics:
        value_str = execution['perf_output'].get(metric, '0')
        value = float(value_str.split()[0].replace(',', ''))
        metrics_data[metric].append(value)

metric_translations = {
    'task_clock': 'Tempo de Tarefa',
    'cycles': 'Ciclos',
    'instructions': 'Instruções',
    'branches': 'Ramificações',
    'branch_misses': 'Falhas de Ramificação',
    'l1_dcache_loads': 'Carregamentos L1 DCache',
    'l1_dcache_load_misses': 'Falhas de Carregamento L1 DCache'
}

# Cálculo das métricas adicionais
def calculate_metrics(execution, real_time, user_time, sys_time):
    instructions = float(execution['perf_output']['instructions'].split()[0].replace(
    cycles = float(execution['perf_output']['cycles'].split()[0].replace(',', ''))

    # Assumindo frequência do clock em Hz (ex: 2.5GHz = 2.5 * 10^9 Hz)
    clock_frequency = 2.5 * 10**9

    # Tempo de CPU
    cpu_time = cycles / clock_frequency
```

```
    # Percentual do tempo total gasto pela CPU no programa
    total_cpu_time = user_time + sys_time
    percent_cpu_time = total_cpu_time / real_time

    # MIPS
    mips = (instructions / total_cpu_time) / 10**6

    # Assumindo um número fictício de operações de ponto flutuante
    floating_point_operations = 10**6
    mflops = (floating_point_operations / total_cpu_time) / 10**6

    # Performance
    performance = 1 / real_time

    return percent_cpu_time, mips, mflops, cpu_time, performance

# Calcular e exibir métricas para cada execução
for i, execution in enumerate(executions):
    real_time = real_times[i]
    user_time = user_times[i]
    sys_time = sys_times[i]
    percent_cpu_time, mips, mflops, cpu_time, performance = calculate_metrics(executi

    print(f"Métricas para {names[i]}:")
    print(f"Percentual do tempo total gasto pela CPU: {percent_cpu_time:.2f}")
    print(f"MIPS: {mips:.2f}")
    print(f"MFLOPS: {mflops:.2f}")
    print(f"Tempo de CPU: {cpu_time:.6f} segundos")
    print(f"Performance: {performance:.6f}")
    print()

factorial_indices = [i for i, name in enumerate(names) if 'fatorial' in name]
other_indices = [i for i, name in enumerate(names) if 'fatorial' not in name]

factorial_names = [names[i] for i in factorial_indices]
factorial_real_times = [real_times[i] for i in factorial_indices]
factorial_user_times = [user_times[i] for i in factorial_indices]
factorial_sys_times = [sys_times[i] for i in factorial_indices]

other_names = [names[i] for i in other_indices]
other_real_times = [real_times[i] for i in other_indices]
other_user_times = [user_times[i] for i in other_indices]
other_sys_times = [sys_times[i] for i in other_indices]

plt.figure(figsize=(10, 5))
plt.barh(factorial_names, factorial_real_times, color='lightcoral', label='Tempo Real
plt.barh(factorial_names, factorial_user_times, color='skyblue', label='Tempo de Usuá
plt.barh(factorial_names, factorial_sys_times, color='lightgreen', label='Tempo de Si
```

```
plt.xlabel('Tempo (segundos)')
plt.title('Comparação dos Tempos de Execução para Fatorial')
plt.legend()
plt.tight_layout()
plt.savefig('resultados/tempo_execucao_fatorial.png')
plt.close()

plt.figure(figsize=(10, 5))
plt.barh(other_names, other_real_times, color='lightcoral', label='Tempo Real')
plt.barh(other_names, other_user_times, color='skyblue', label='Tempo de Usuário', le
plt.barh(other_names, other_sys_times, color='lightgreen', label='Tempo de Sistema',
plt.xlabel('Tempo (segundos)')
plt.title('Comparação dos Tempos de Execução para Outras Funções')
plt.legend()
plt.tight_layout()
plt.savefig('resultados/tempo_execucao_outras.png')
plt.close()

for metric, values in metrics_data.items():
    plt.figure(figsize=(10, 5))
    plt.barh(names, values, color='skyblue')
    plt.xlabel(metric_translations[metric])
    plt.title(f'Comparação de {metric_translations[metric]} entre Execuções')
    plt.tight_layout()
    plt.savefig(f'resultados/{metric}.png')
    plt.close()
```

2 Resultados e Discussão

Os gráficos a seguir apresentam os tempos de execução dos algoritmos recursivo e iterativo para o cálculo do fatorial, bem como métricas de desempenho coletadas com a ferramenta *perf*.

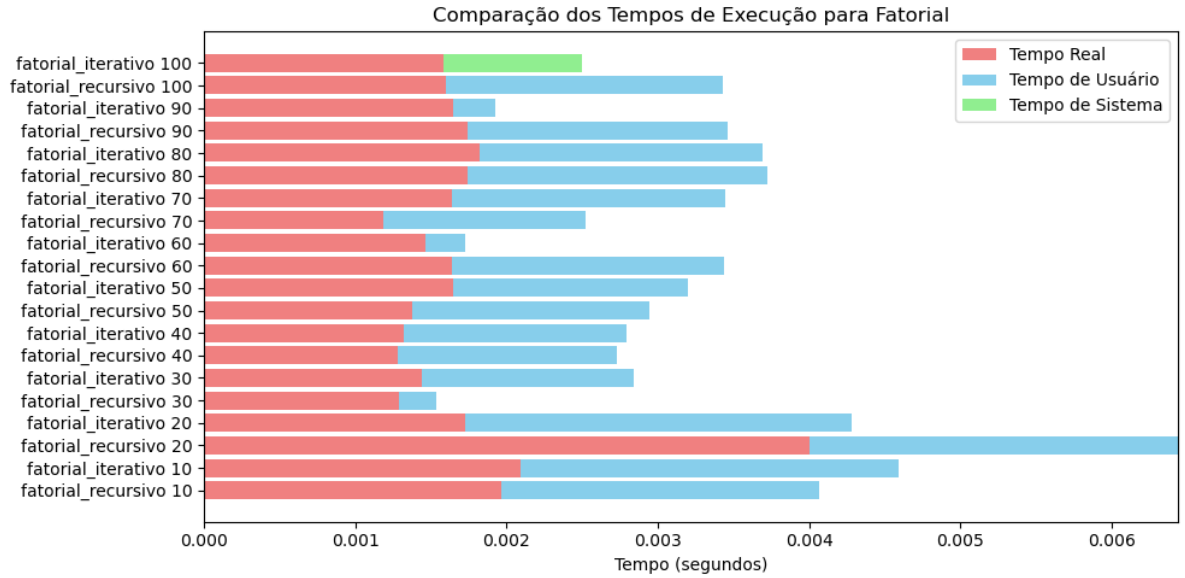

Figura 2: Tempo de Execução dos Algoritmos de Fatorial (perf)

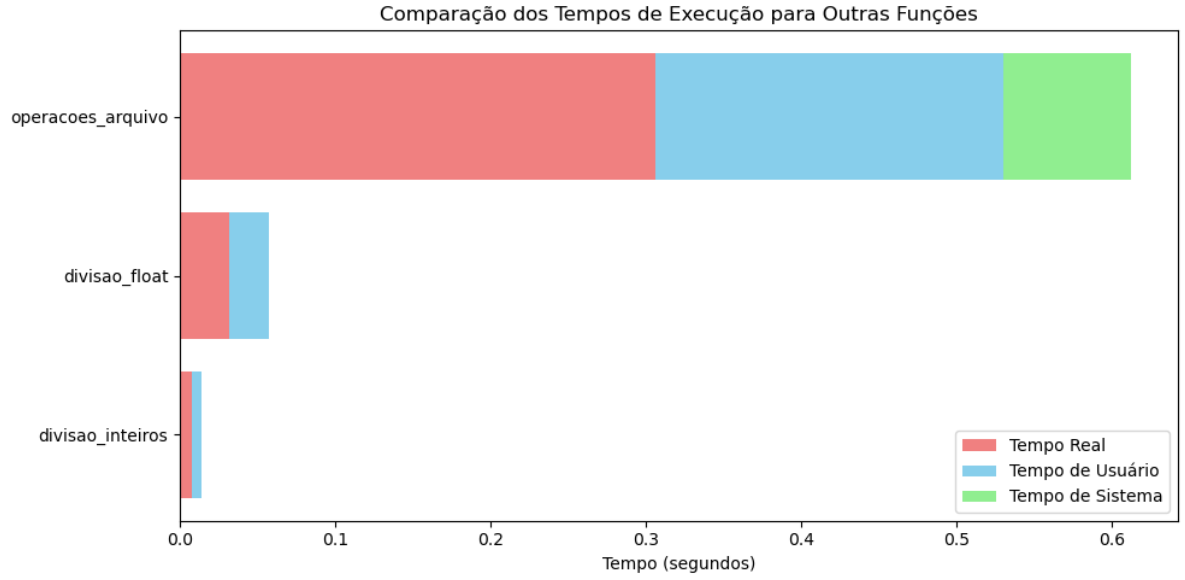

Figura 3: Tempo de Execução de Outras Funções

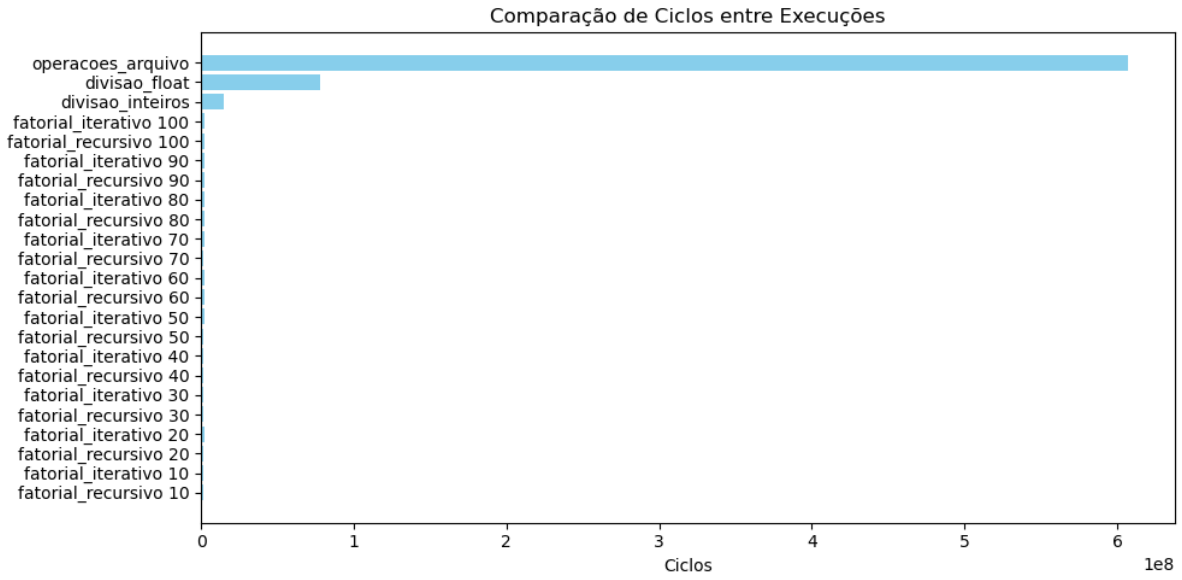

Figura 4: Ciclos de CPU dos Algoritmos

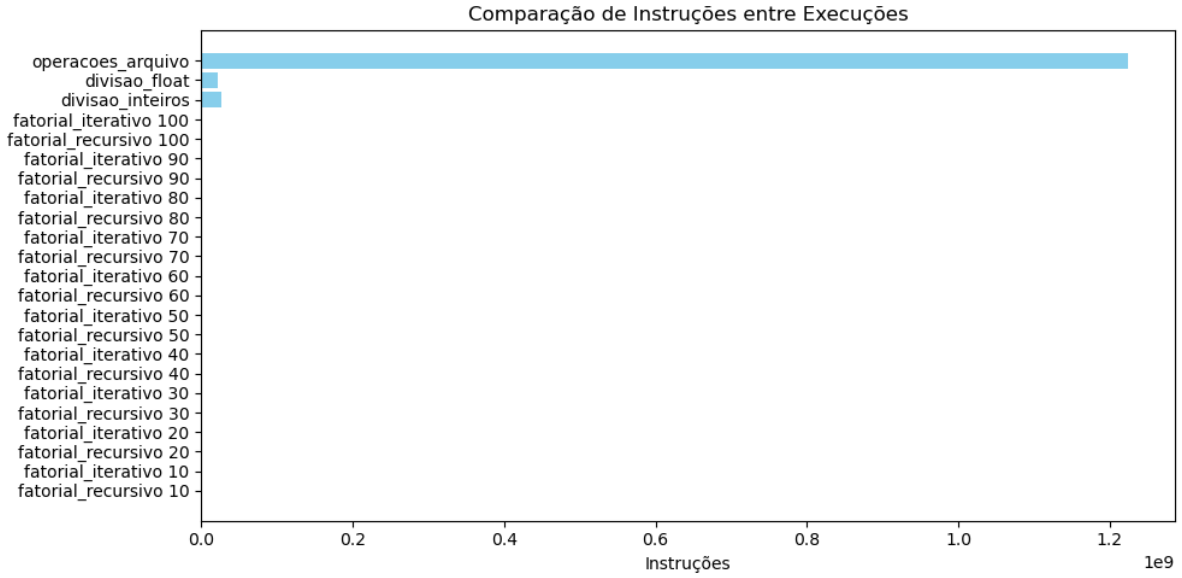

Figura 5: Instruções de CPU dos Algoritmos

Os resultados mostram que o algoritmo iterativo tem um desempenho melhor em termos de tempo de execução e ciclos de CPU quando comparado ao algoritmo recursivo. As operações de divisão de inteiros e floats apresentaram desempenho similar, enquanto a operação de escrita em arquivo adicionou um overhead significativo devido ao I/O. A análise detalhada das métricas forneceu uma visão abrangente do comportamento dos algoritmos.

3 Análise das Métricas

Nesta seção, apresentamos as métricas de desempenho detalhadas para diferentes algoritmos e entradas. Cada métrica é explicada abaixo com suas respectivas fórmulas:

- **Percentual do Tempo Total Gasto pela CPU**: Representa a porcentagem do tempo total de execução que a CPU utilizou para processar o algoritmo. É calculado como:

$$\text{Percentual do Tempo CPU} = \frac{\text{Tempo de Usuário} + \text{Tempo de Sistema}}{\text{Tempo Real}} \tag{1}$$

- **MIPS (Milhões de Instruções por Segundo)**: Mede o número de instruções executadas pela CPU por segundo. É calculado como:

$$\text{MIPS} = \frac{\text{Número de Instruções}}{\text{Tempo de CPU} \times 10^6} \tag{2}$$

- **MFLOPS (Milhões de Operações de Ponto Flutuante por Segundo)**: Mede o número de operações de ponto flutuante executadas pela CPU por segundo. É calculado como:

$$\text{MFLOPS} = \frac{\text{Número de Operações de Ponto Flutuante}}{\text{Tempo de CPU} \times 10^6} \tag{3}$$

- **Tempo de CPU**: Tempo total de uso da CPU para executar o algoritmo, calculado com base no número de ciclos de clock e a frequência do clock:

$$\text{Tempo de CPU} = \frac{\text{Número de Ciclos de Clock}}{\text{Frequência do Clock}} \tag{4}$$

- **Performance**: Uma medida geral de desempenho, calculada como o inverso do tempo de execução:

$$\text{Performance} = \frac{1}{\text{Tempo de Execução}} \tag{5}$$

3.1 Métricas Detalhadas

- **Métricas para fatorial_recursivo 10**:
  - Percentual do tempo total gasto pela CPU: 1.07
  - MIPS: 328.94
  - MFLOPS: 475.06
  - Tempo de CPU: 0.000585 segundos
  - Performance: 510.041701
- **Métricas para fatorial_iterativo 10**:
  - Percentual do tempo total gasto pela CPU: 1.19
  - MIPS: 282.64

  - MFLOPS: 400.00
  - Tempo de CPU: 0.000618 segundos
  - Performance: 477.432027

- **Métricas para fatorial_recursivo 20**:
  - Percentual do tempo total gasto pela CPU: 0.61
  - MIPS: 289.25
  - MFLOPS: 410.68
  - Tempo de CPU: 0.000677 segundos
  - Performance: 249.671869

- **Métricas para fatorial_iterativo 20**:
  - Percentual do tempo total gasto pela CPU: 1.48
  - MIPS: 278.49
  - MFLOPS: 391.85
  - Tempo de CPU: 0.000690 segundos
  - Performance: 578.603921

- **Métricas para fatorial_recursivo 30**:
  - Percentual do tempo total gasto pela CPU: 0.19
  - MIPS: 2995.12
  - MFLOPS: 4166.67
  - Tempo de CPU: 0.000603 segundos
  - Performance: 774.745167

- **Métricas para fatorial_iterativo 30**:
  - Percentual do tempo total gasto pela CPU: 0.97
  - MIPS: 513.58
  - MFLOPS: 714.29
  - Tempo de CPU: 0.000643 segundos
  - Performance: 695.476067

- **Métricas para fatorial_recursivo 40**:
  - Percentual do tempo total gasto pela CPU: 1.14
  - MIPS: 523.93
  - MFLOPS: 688.71
  - Tempo de CPU: 0.000625 segundos
  - Performance: 781.710476

- **Métricas para fatorial_iterativo 40**:

  - Percentual do tempo total gasto pela CPU: 1.12
  - MIPS: 512.61
  - MFLOPS: 676.59
  - Tempo de CPU: 0.000642 segundos
  - Performance: 759.330271

- **Métricas para fatorial_recursivo 50**:
  - Percentual do tempo total gasto pela CPU: 1.15
  - MIPS: 503.98
  - MFLOPS: 634.92
  - Tempo de CPU: 0.000664 segundos
  - Performance: 728.938949

- **Métricas para fatorial_iterativo 50**:
  - Percentual do tempo total gasto pela CPU: 0.94
  - MIPS: 506.31
  - MFLOPS: 645.16
  - Tempo de CPU: 0.000688 segundos
  - Performance: 606.211975

- **Métricas para fatorial_recursivo 60**:
  - Percentual do tempo total gasto pela CPU: 1.10
  - MIPS: 463.51
  - MFLOPS: 555.86
  - Tempo de CPU: 0.000781 segundos
  - Performance: 610.654579

- **Métricas para fatorial_iterativo 60**:
  - Percentual do tempo total gasto pela CPU: 0.18
  - MIPS: 3214.19
  - MFLOPS: 3875.97
  - Tempo de CPU: 0.000681 segundos
  - Performance: 682.795666

- **Métricas para fatorial_recursivo 70**:
  - Percentual do tempo total gasto pela CPU: 1.13
  - MIPS: 659.18
  - MFLOPS: 745.71
  - Tempo de CPU: 0.000631 segundos

  - Performance: 844.712312

- **Métricas para fatorial_iterativo 70**:
  - Percentual do tempo total gasto pela CPU: 1.10
  - MIPS: 492.04
  - MFLOPS: 553.10
  - Tempo de CPU: 0.000864 segundos
  - Performance: 611.065044

- **Métricas para fatorial_recursivo 80**:
  - Percentual do tempo total gasto pela CPU: 1.14
  - MIPS: 483.83
  - MFLOPS: 505.05
  - Tempo de CPU: 0.000808 segundos
  - Performance: 573.493476

- **Métricas para fatorial_iterativo 80**:
  - Percentual do tempo total gasto pela CPU: 1.03
  - MIPS: 503.15
  - MFLOPS: 533.90
  - Tempo de CPU: 0.000878 segundos
  - Performance: 549.992383

- **Métricas para fatorial_recursivo 90**:
  - Percentual do tempo total gasto pela CPU: 0.98
  - MIPS: 591.03
  - MFLOPS: 583.43
  - Tempo de CPU: 0.000797 segundos
  - Performance: 573.065903

- **Métricas para fatorial_iterativo 90**:
  - Percentual do tempo total gasto pela CPU: 0.17
  - MIPS: 3672.06
  - MFLOPS: 3623.19
  - Tempo de CPU: 0.000854 segundos
  - Performance: 607.870217

- **Métricas para fatorial_recursivo 100**:
  - Percentual do tempo total gasto pela CPU: 1.15

  - MIPS: 595.57
  - MFLOPS: 545.55
  - Tempo de CPU: 0.000809 segundos
  - Performance: 625.671815

- **Métricas para fatorial_iterativo 100**:
  - Percentual do tempo total gasto pela CPU: 0.58
  - MIPS: 1196.09
  - MFLOPS: 1096.49
  - Tempo de CPU: 0.000785 segundos
  - Performance: 631.041831

- **Métricas para divisao_inteiros**:
  - Percentual do tempo total gasto pela CPU: 0.85
  - MIPS: 4265.53
  - MFLOPS: 160.49
  - Tempo de CPU: 0.005984 segundos
  - Performance: 136.773927

- **Métricas para divisao_float**:
  - Percentual do tempo total gasto pela CPU: 0.82
  - MIPS: 832.82
  - MFLOPS: 38.69
  - Tempo de CPU: 0.031143 segundos
  - Performance: 31.678046

- **Métricas para operacoes_arquivo**:
  - Percentual do tempo total gasto pela CPU: 1.00
  - MIPS: 4002.54
  - MFLOPS: 3.27
  - Tempo de CPU: 0.242996 segundos
  - Performance: 3.268903

4 Conclusão

A análise de desempenho realizada revelou que, para o cálculo do fatorial de números relativamente grandes, o método iterativo tende a ser menos discrepante em termos de tempo de execução e uso de ciclos de CPU. As operações de divisão mostraram-se eficientes, enquanto as operações de escrita em arquivo apresentaram maior overhead devido ao I/O. A ferramenta *perf* foi essencial para coletar métricas detalhadas de desempenho, possibilitando uma análise mais aprofundada do que o *time*.