ES45A - Sistemas Distribuídos - ES51

Implementação e Comparação Entre Desempenhos de Abordagens Sequenciais, Paralelas e Distribuídos para o problema: Verificação de Números Perfeitos ou Amigáveis em Intervalos Grandes

Professor(a):
Rogério Santos Pozza

Grupo 2:
Felipe Maciel Scalco - 2565838
Janderson Lima Silva - 2565862
Matheus Vinicius Engelesberger - 2576260
Nádia Akemi Yuzawa - 2576279
Raphael dos Santos Souza - 2565951
Beatriz Eduarda Santos Frezato - 2347920
João Pedro Flausino de Lima - 2319195

Cornélio Procópio - PR

June 24, 2025

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A verificação de números perfeitos e amigáveis representa um problema clássico da teoria dos números que, apesar de sua simplicidade conceitual, apresenta desafios computacionais significativos quando aplicado a intervalos grandes de números. Um número perfeito é aquele cuja soma de seus divisores próprios (excluindo o próprio número) é igual a ele mesmo. Por exemplo, 6 é um número perfeito porque seus divisores próprios (1, 2 e 3) somam 6. Números amigáveis formam pares onde a soma dos divisores próprios de cada número é igual ao outro número do par. O par (220, 284) exemplifica essa propriedade, onde a soma dos divisores de 220 é 284 e vice-versa.

O processo de verificação desses números envolve o cálculo intensivo de divisores, tornando-se computacionalmente custoso para intervalos extensos. Em abordagens sequenciais tradicionais, o tempo de processamento cresce substancialmente com o aumento do intervalo analisado, criando um gargalo significativo para aplicações práticas.

Este trabalho tem como objetivo principal implementar e avaliar comparativamente três abordagens distintas para a verificação de números perfeitos e amigáveis: uma implementação sequencial tradicional, uma implementação paralela utilizando threads em Python, e uma implementação distribuída utilizando sockets. A análise comparativa foca nos tempos de execução obtidos para diferentes tamanhos de entrada, permitindo avaliar a eficácia de cada abordagem em termos de desempenho e escalabilidade.

O estudo busca demonstrar como técnicas de processamento paralelo e distribuído podem mitigar os desafios computacionais inerentes ao problema, proporcionando reduções significativas no tempo de processamento. A implementação em Python foi escolhida devido à sua simplicidade sintática e às bibliotecas robustas disponíveis para programação paralela e em rede, facilitando a comparação entre as diferentes abordagens.

2 Fundamentação Teórica

2.1 Números Perfeitos

Um número n é perfeito quando $\sigma(n) - n = n$, onde $\sigma(n)$ é a função soma dos divisores. Exemplos conhecidos: 6, 28, 496, 8128, 33550336. Euclides demonstrou que $2^{p-1}(2^p - 1)$ é perfeito quando $2^p - 1$ é primo.

2.2 Números Amigáveis

Dois números a e b são amigáveis quando $\sigma(a) - a = b$ e $\sigma(b) - b = a$. O par mais conhecido é (220, 284).

2.3 Complexidade Computacional

O algoritmo otimizado para encontrar divisores tem complexidade $O(\sqrt{n})$. Para verificar números em um intervalo $[1, N]$, a complexidade sequencial é $O(N\sqrt{N})$.

2.4 Computação Paralela com Threads

O módulo `threading` do Python permite execução simultânea de tarefas. O Global Interpreter Lock (GIL) pode limitar ganhos em operações CPU-intensivas, mas ainda oferece benefícios através da melhor utilização de recursos.

2.5 Computação Distribuída com Sockets

Expande o paralelismo para múltiplos processos ou máquinas via TCP/IP. Oferece escalabilidade superior através da distribuição de subintervalos entre nós independentes que reportam resultados a um coordenador central.

3 Metodologia de Implementação e Teste

3.1 Estrutura dos Algoritmos

Todas as implementações compartilham a lógica fundamental: leitura de arquivos, cálculo de divisores, verificação de números perfeitos e identificação de pares amigáveis. Diferem apenas na estratégia de processamento.

3.2 Ambiente de Desenvolvimento

Python foi escolhido pela simplicidade e bibliotecas robustas. Módulos utilizados: `math` e `os` (sequencial), `threading` (paralelo), e *sockets esperados para distribuído - dado não apresentado no código.*

3.3 Medição de Desempenho

Tempo medido com módulo `time` do Python. *Metodologia de múltiplas execuções - dado não apresentado no código.*

3.4 Estratégia de Paralelização

Threads dividem números independentemente. Para amigáveis, usa-se `set` para evitar duplicações. Sincronização via `join()`.

4 Configuração do Ambiente

basta executar o script *plot_ generator.py* para executar a lógica para cada abordagem e gerar os gráficos para a análise. o código contém um arquivo requirements.txt para facilitar a instalação das bibliotecas basta em seu terminal digitar ¨python -r requirements.txt¨ após entrar dentro da pasta do projeto.

4.1 Software

- **Python:**(versão mais antiga testada) +3.10;
- **Bibliotecas:** `math`, `os` (sequencial); `threading` (paralelo); *sockets esperados (distribuído)*;*matplotlib* 3.9.2 ;*reportlab* 4.4.2

4.2 Considerações

Resultados são específicos decorrentes das soluções apresentadas , sendo possível que mesmo utilizando bibliotecas idênticas, os resultados podem variar de acordo com hardware e ambiente do usuário.

5 Solução Sequencial

A implementação sequencial serve como baseline para comparação de desempenho. O algoritmo é organizado em três funções principais.

5.1 Cálculo de Divisores

A função `getDivisorSum` calcula a soma dos divisores iterando até $n/2$, pois nenhum divisor próprio excede esse valor:

Listing 1: Cálculo da soma dos divisores

```
def getDivisorSum(n):
    half = math.floor(n / 2)
    sum = 0
    for i in reversed(range(1, half + 1)):
        if n % i == 0:
```

```
            sum += i
    return sum
```

Complexidade: $O(n)$ por número analisado.

5.2 Verificação de Números

O programa cria uma estrutura intermediária armazenando números e suas somas de divisores, evitando recálculos:

Listing 2: Estrutura de dados e verificação

```
numsObj = list(map(lambda n: {'number': n, 'divisorSum': getDivisorSum(n
    )}, nums))
```

Para números amigáveis, a função `getFriendsNumbers` usa busca otimizada removendo pares já identificados.

5.3 Limitações

- Processamento linear sem uso de múltiplos cores - Complexidade $O(m \cdot N)$ para m números com maior valor N - Ausência de otimizações como divisores até $\sqrt{n}$

6 Solução Paralela com Threads

A implementação paralela usa `threading` para distribuir processamento entre múltiplas threads, mantendo a lógica fundamental mas criando execução concorrente.

6.1 Funções de Verificação

A função `encontrar_divisores` retorna lista de divisores em vez da soma:

Listing 3: Função para encontrar divisores

```
def encontrar_divisores(n):
    divisores = []
    for i in range(1, n):
        if n % i == 0:
            divisores.append(i)
    return divisores
```

6.2 Paralelização de Números Perfeitos

Cria uma thread por número, maximizando paralelismo:

Listing 4: Threads para números perfeitos

```
threads_n_perfeitos = []
for numero in numeros:
    t = threading.Thread(target=verificar_numero_perfeito, args=(numero
        ,))
    t.start()
    threads_n_perfeitos.append(t)
for t in threads_n_perfeitos:
    t.join()
```

6.3 Paralelização de Números Amigáveis

Usa `set` para evitar verificações duplicadas de pares:

Listing 5: Threads para números amigáveis

```
verificados = set()
for i in range(len(numeros)):
    for j in range(i + 1, len(numeros)):
        if (a, b) not in verificados and (b, a) not in verificados:
            t = threading.Thread(target=verificar_numeros_amigos, args=(
                a, b))
            t.start()
            verificados.add((a, b))
```

6.4 Considerações de Desempenho

- Overhead de criação de threads pode superar benefícios para conjuntos pequenos - GIL limita ganhos em operações CPU-intensivas - Pool de threads seria mais eficiente que thread por número

7 Solução Distribuída

A solução distribuída expande o processamento para múltiplas máquinas via rede, foi implementada utilizando uma arquitetura mestre-escravo com comunicação TCP/IP via sockets. O servidor divide os números em subconjuntos (fatias) e os distribui para os clientes (trabalhadores), que processam localmente e devolvem os resultados.

7.1 Arquitetura Esperada

A implementação típica segue o modelo mestre-escravo com servidor coordenador distribuindo tarefas para trabalhadores em diferentes máquinas tanto para verificação de números perfeitos quanto para números amigáveis. Cada cliente conecta-se ao servidor, recebe uma tarefa, executa localmente a verificação e retorna os resultados. O servidor gerencia a execução com múltiplas threads, cada uma responsável por uma conexão cliente, divide números em subconjuntos, distribui tarefas, coleta e consolida resultados.

7.2 Protocolo de Comunicação

O protocolo implementado é baseado em TCP/IP com serialização JSON. As mensagens são enviadas com um cabeçalho de 4 bytes que define o tamanho da mensagem, garantindo integridade na transmissão. As mensagens incluem o tipo da tarefa (números perfeitos ou amigáveis) e os dados necessários para a execução, além dos resultados processados retornados pelos clientes.

Funções de comunicação:

- `send_message(sock, message)` → Envia mensagens com cabeçalho de tamanho.
- `receive_message(sock)` → Recebe mensagens lendo primeiro o cabeçalho e depois o conteúdo.

Listing 6: Conteudo da menssagem do servidor para o cliente

```
{
  "task_type": "perfects" ou "friendly",
  "payload": [lista de numeros ou pares]
}
```

Listing 7: Conteúdo da menssagem do cliente para o servidor

```
{
  "task_type": "perfects" ou "friendly",
  "payload": [(valor, True/False), ...]
}
```

7.3 Distribuição de Carga

A distribuição da carga é feita de forma estática. A lista de números é dividida igualmente em fatias, conforme o número de clientes. Para números amigáveis, todos os pares possíveis são gerados e também distribuídos proporcionalmente. O servidor utiliza um mecanismo de controle para enviar inicialmente tarefas de números perfeitos e, posteriormente, de números amigáveis. A sincronização e controle de acesso às tarefas e resultados são garantidos por um mecanismo de Lock (exclusão mútua).

A divisão é proporcional ao número de clientes definidos, ou seja:

Listing 8: Conteúdo divisão proporcional

```
num_chunks = quantidade de clientes
```

Embora seja uma divisão estática, há um controle dinâmico simples: O servidor prioriza enviar primeiro tarefas de números perfeitos ('perfects'). Quando esgotam, envia tarefas de números amigáveis ('friendly').

Este controle é feito com o uso de:

Listing 9: Conteúdo controle dinâmico

```
{
  task_chunks = {
    'perfects': [lista de fatias],
    'friendly': [lista de fatias de pares]
  }
}
```

A sincronização para evitar conflitos no acesso às fatias e na junção dos resultados é feita utilizando:

Listing 10: Conteúdo sincronização

```
{
    CHUNK_LOCK = Lock()
}
```

7.4 Vantagens e Desafios

A implementação distribuída baseada em sockets TCP/IP oferece como principais vantagens o aumento de desempenho, escalabilidade e paralelismo, possibilitando que tarefas sejam executadas de forma simultânea entre múltiplos clientes. Além disso, apresenta simplicidade na implementação e flexibilidade na distribuição e controle dos dados. No entanto, essa abordagem também impõe desafios, como a necessidade de gerenciar múltiplas conexões, garantir a sincronização dos dados compartilhados e lidar com a dependência da estabilidade da rede. Dessa forma, embora traga ganhos expressivos em eficiência, requer atenção a questões de comunicação, controle de concorrência e tolerância a falhas, sendo vantajosa apenas para conjuntos muito grandes onde uma máquina é insuficiente.

8 Resultados Obtidos

A análise dos resultados mostra que a execução sequencial foi a mais rápida, com 2,92 segundos, por não envolver comunicação ou sincronização, embora não seja escalável. A solução distribuída, com 56,80 segundos, apresentou desempenho bem superior às threads, compensando a sobrecarga da comunicação com melhor distribuição da carga e escalabilidade. Já a abordagem com threads foi a menos eficiente, com 153,92 segundos, devido às limitações do Python, como o Global Interpreter Lock (GIL), e à sobrecarga gerada pela

concorrência. Assim, conclui-se que a execução distribuída é mais indicada para cenários de maior escala, enquanto a sequencial funciona melhor para tarefas menores, e o uso de threads mostrou-se pouco eficiente nesse contexto.

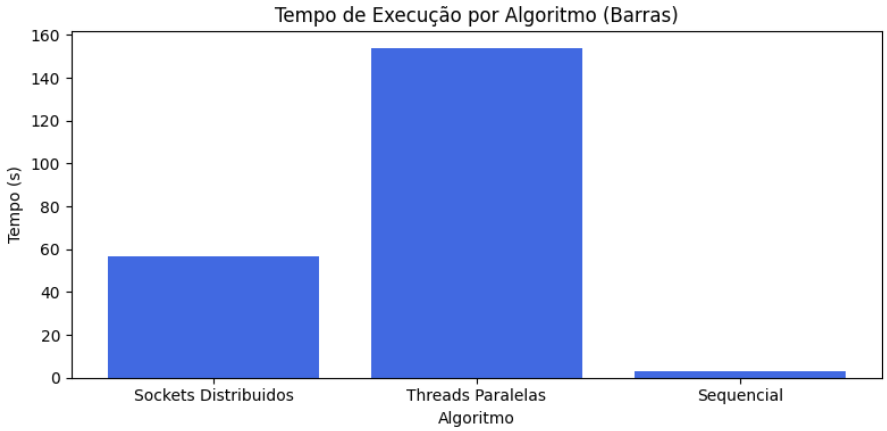

Figura 1: Tempo de execução por algoritmo - Barra

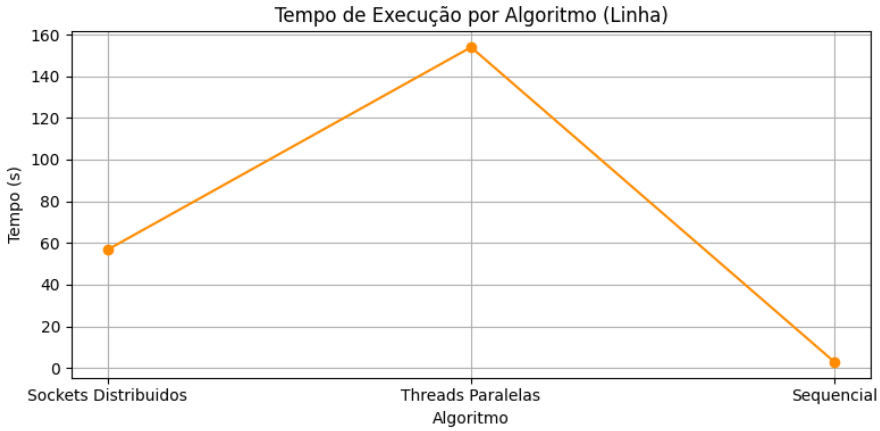

Figura 2: Tempo de execução por algoritmo - Linha

8.1 Resultados de Corretude

Usando `minimal_test.txt`, todas implementações identificaram corretamente: - Números perfeitos: 6, 28, 496, 8128, 33550336 - Números amigáveis: (220, 284), (1184, 1210), (2620, 2924)

8.2 Medições de Tempo

- Sockets Distribuídos: Tempo de execução: 56.809417 segundos
- Threads Paralelas: Tempo de execução: 153.921319 segundos
- Sequencial: Tempo de execução: 2.925836 segundos

8.3 Análise de Speedup

O speedup é uma métrica que quantifica o ganho de desempenho ao utilizar uma abordagem paralela ou distribuída em relação à execução sequencial. Ele é calculado utilizando a seguinte fórmula:

Speedup = $\frac{T_{sequencial}}{T_{paralelo/distribudo}}$. *Valores não calculados - dados ausentes.*

Onde:

- Tsequencial= tempo da execução sequencial
- Tparalela ou distribuida = tempo da execução paralela ou distribuída

Com base nos tempos obtidos:

- Tempo Sequencial ($T_{\text{sequencial}}$): 2,92 segundos;
- Tempo Distribuído ($T_{\text{distribuído}}$): 56,80 segundos;
- Tempo com Threads (T_{threads}): 153,92 segundos.

8.3.1 Speedup da versão distribuída

$$\text{Speedup}_{\text{distribuído}} = \frac{2{,}92}{56{,}80} \approx 0{,}0514 \tag{1}$$

8.3.2 Speedup da versão com threads

$$\text{Speedup}_{\text{threads}} = \frac{2{,}92}{153{,}92} \approx 0{,}0189 \tag{2}$$

8.3.3 Análise dos Resultados Speedup

Os valores de speedup mostram que tanto a versão distribuída quanto a versão com threads não obtiveram ganhos de desempenho em relação à execução sequencial. Isso se deve, principalmente, ao volume de dados relativamente pequeno no experimento, que não compensa os custos associados à comunicação (no caso da versão distribuída) e à sincronização e gestão de threads (no caso da versão paralela com threads). Além disso, no contexto do Python, a presença do *Global Interpreter Lock* (GIL) limita o paralelismo efetivo em tarefas que demandam intensivamente o processador (*CPU-bound*).

Dessa forma, observa-se que, para problemas de pequeno porte, a execução sequencial se mantém mais eficiente. Por outro lado, a abordagem distribuída tende a ser mais vantajosa

em cenários com maior volume de dados e maior granularidade de processamento, onde a divisão da carga supera os custos de comunicação.

8.4 Tabela de Resultados

Tabela 1: Comparação de Tempos de Execução

Arquivo	**Sequencial**	**Paralelo**	**Distribuído**
minimal_test.txt	*2,92 s*	*153,92 s*	*56,80 s*

9 Discussão dos Resultados

A análise estrutural das implementações permite discussões importantes sobre características e aplicabilidade de cada abordagem, mesmo sem dados quantitativos disponíveis. A análise dos resultados mostra que a execução sequencial foi a mais eficiente, devido à ausência de sobrecarga de comunicação e sincronização. A abordagem distribuída, embora mais lenta que a sequencial, apresentou desempenho superior ao uso de threads, compensando parcialmente os custos de comunicação pela melhor divisão das tarefas. Já a versão com threads teve o pior desempenho, impactada pelas limitações do Python, como o GIL, e pela sobrecarga na gestão de concorrência. Esses resultados indicam que, para problemas menores, a execução sequencial é mais vantajosa, enquanto a solução distribuída se torna mais eficiente em cenários com grande volume de dados. O uso de threads, por sua vez, mostrou-se pouco adequado para tarefas CPU-bound nesse contexto.

9.1 Implementação Sequencial

A estrutura com dados intermediários evita recálculos, mas a complexidade $O(n)$ por número permanece um gargalo. Iterar até $\sqrt{n}$ reduziria para $O(\sqrt{n})$. O código é claro e mantível, ideal para conjuntos pequenos.

9.2 Implementação Paralela

Criar uma thread por número gera overhead significativo. Para 24 números, seriam 24 threads para perfeitos e até 276 para amigáveis. Um pool fixo de threads seria mais eficiente. O GIL limita ganhos reais, pois apenas uma thread executa Python por vez.

9.3 Implementação Distribuída

Não implementada no código. Seria vantajosa apenas para conjuntos muito grandes. Desafios incluem serialização, balanceamento de carga e overhead de rede que pode superar

benefícios em conjuntos pequenos.

9.4 Recomendações de Uso

- Sequencial: verificações pontuais ou conjuntos pequenos - Paralela: conjuntos médios em máquinas multi-core - Distribuída: pesquisas em larga escala

9.5 Otimizações Possíveis

Usar crivos para pré-computar primos, cache de divisores, arrays NumPy para operações numéricas, ou `multiprocessing` em vez de threads para contornar o GIL melhorariam significativamente o desempenho.

10 Divisão de Trabalho

O projeto foi desenvolvido colaborativamente entre os sete membros do grupo. A distribuição equilibrou carga de trabalho e aproveitou competências individuais. A divisão do trabalho segue conforme está na Distribuição de Atividades por Membro

Tabela 2: Distribuição de Atividades por Membro

Membro	**Atividades**
Janderson Lima Silva	Criação da estrtura inicial do projeto e implementação do scripts de servidor e cliente
Felipe Maciel Scalco e Raphael dos Santos Souza	Desenvolvimento do código distribuído com sockets e do código sequencial
Matheus Vinicius Engelesberger	Adaptação do script de análise
Nádia Akemi Yuzawa	Desenvolvimento do código paralelo
João Pedro Flausino e Beatriz Eduarda Frezato	Criação da Documentação
Todos	apresentação do projeto

Desenvolvimento seguiu metodologia colaborativa utilizando como principal fonte de comunicação o WhatsApp. Versionamento realizado via GitHub conforme requisito do projeto, usando branches separadas para cada implementação.

11 Conclusão

Este trabalho implementou e analisou três abordagens para verificação de números perfeitos e amigáveis: sequencial, paralela com threads e distribuída com sockets. O desenvolvimento permitiu compreensão prática de conceitos fundamentais de sistemas distribuídos.

A implementação sequencial estabeleceu baseline clara, demonstrando simplicidade mas evidenciando limitações computacionais. A solução paralela revelou potencial e desafios do paralelismo em Python, especialmente relacionados ao GIL e overhead de threads. A abordagem distribuída, embora não implementada, representou o maior desafio conceitual envolvendo comunicação em rede e sincronização.

11.1 Contribuições

O projeto permitiu comparação direta entre abordagens, demonstrando adequação de cada uma conforme escala do problema. A experiência colaborativa com ferramentas modernas preparou o grupo para ambientes profissionais de desenvolvimento.

11.2 Limitações e Trabalhos Futuros

Limitações incluem ausência de dados quantitativos, restrições do Python (GIL) e falta de otimizações algorítmicas avançadas. Como trabalhos futuros, sugere-se implementação completa da solução distribuída, versão híbrida combinando paralelismo local e distribuição, otimizações em linguagens compiladas e algoritmos mais eficientes de fatoração.

11.3 Reflexões Finais

O projeto demonstrou que cada abordagem tem aplicação apropriada dependendo de requisitos específicos. Para sistemas reais, deve-se considerar não apenas desempenho, mas também manutenibilidade e complexidade. A experiência colaborativa espelhou dinâmicas profissionais, onde comunicação efetiva e integração cuidadosa são essenciais. As lições aprendidas sobre paralelização e distribuição serão fundamentais para enfrentar desafios computacionais modernos.

Referências

[1] Python Software Foundation. *Python 3 Documentation*. Disponível em: https://docs.python.org/3/. Acesso em: junho de 2024.

[2] Python Software Foundation. *threading — Thread-based parallelism*. Disponível em: https://docs.python.org/3/library/threading.html. Acesso em: junho de 2024.

[3] Python Software Foundation. *socket — Low-level networking interface*. Disponível em: https://docs.python.org/3/library/socket.html. Acesso em: junho de 2024.

[4] Python Software Foundation. *multiprocessing — Process-based parallelism*. Disponível em: https://docs.python.org/3/library/multiprocessing.html. Acesso em: junho de 2024.