Computação Escalável: Análise Comparativa de Modelos de Paralelismo em dados meteorológicos artificiais

AS - Computação Escalável

João Felipe Vilas Boas

6 de Julho de 2025

Resumo

Este relatório detalha o desenvolvimento e os resultados de um experimento prático que compara três modelos de paralelismo — multiprocessamento local, message broker com workers e Apache Spark — aplicados a um problema de processamento de dados meteorológicos. O objetivo principal é analisar as vantagens, desvantagens e características de desempenho de cada abordagem em diferentes cenários de carga, fornecendo uma base empírica para a escolha da arquitetura mais adequada.

Conteúdo

1 Introdução

1.1 Contexto e Motivação

A rede ClimaData coleta dados de estações meteorológicas distribuídas geograficamente. São gerados grandes volumes de informações sobre temperatura, umidade e pressão atmosférica a cada minuto. Você foi encarregado de criar um sistema que receba os dados do dia anterior e compute algumas métricas. Embora você saiba qual é a arquitetura ideal, pediram para você produzir um experimento que comprove dentre três abordagens qual é mais adequada. (Extraída do documento do trabalho)

1.2 Objetivos do Trabalho

O presente trabalho possui os seguintes objetivos específicos:

- **Implementar um gerador de dados** meteorológicos sintéticos que simule a coleta de múltiplas estações.
- **Desenvolver três soluções de processamento paralelo** para o mesmo problema, utilizando (1) multiprocessamento local, (2) um message broker com workers distribuídos, e (3) Apache Spark.
- **Criar uma aplicação de controle e um dashboard** para orquestrar os experimentos e comparar visualmente o desempenho e a corretude das três abordagens.
- **Analisar e discutir os resultados** obtidos, comparando as vantagens e desvantagens de cada modelo de paralelismo.

2 Metodologia

Nesta seção, descrevemos as ferramentas e técnicas utilizadas para construir o ambiente do experimento e calcular as métricas.

2.1 Geração de Dados Sintéticos

Para simular um cenário realista, foi desenvolvido um gerador de dados em Python. Cada evento meteorológico gerado contém os campos obrigatórios: timestamp, ID da estação, região, temperatura, umidade e pressão. O gerador permite configurar o volume total de eventos e o percentual de anomalias a serem introduzidas aleatoriamente, que são persistidas em um arquivo separado para posterior validação da corretude.

2.2 Métricas Analisadas

O sistema foi projetado para computar três métricas distintas, conforme os requisitos:

2.2.1 Percentual de Anomalias por Estação

Para cada evento, uma função de verificação `is_anomalous` avalia os valores dos sensores contra limites pré-definidos. O percentual final é o total de anomalias detectadas em cada sensor, dividido pelo total de eventos daquela estação.

2.2.2 Média Móvel por Região

Uma média móvel de temperatura, umidade e pressão é calculada para cada região, utilizando uma janela deslizante. Os valores identificados como anômalos são excluídos deste cálculo para não distorcer o resultado.

2.2.3 Períodos de Anomalias Múltiplas

Esta métrica contabiliza o número de vezes que uma mesma estação registra anomalias em sensores distintos (por exemplo, temperatura e umidade) dentro de um intervalo de 10 minutos.

2.3 Ambiente de Teste

Todo o ambiente do experimento foi orquestrado utilizando Docker e Docker Compose para garantir a reprodutibilidade e o isolamento dos serviços. A pilha de serviços inclui:

- **RabbitMQ:** Utilizado como message broker na segunda solução.
- **Apache Spark:** Cluster com um master e um worker para a terceira solução.
- **Dashboard:** Container com a aplicação Streamlit e as dependências Python (incluindo o Java para o cliente PySpark).

3 Implementação Detalhada das Métricas

Aqui detalho como é calculada cada métrica em cada modelo. Passaremos novamente em alguns casos quando entrarmos mais em detalhes sobre as abordagens individualmente.

3.1 Métrica 1: Percentual de Anomalias por Estação

Lógica em Python (Multiprocessing e Message Broker): A detecção de anomalias foi centralizada na função `is_anomalous` no módulo `core/metrics.py`. Esta função recebe um dicionário representando um evento e retorna um booleano indicando se é uma anomalia. A lógica é uma série de comparações diretas com limites pré-definidos, que espelham as regras de geração de dados.

```python
# Em core/metrics.py
def is_anomalous(event: dict) -> tuple[bool, str | None]:
    if event['temperature'] >= 45.0 or event['temperature'] <= -10.0:
        return (True, 'temperature')

    if event['humidity'] > 100.0 or event['humidity'] < 0.0:
        return (True, 'humidity')

    if event['pressure'] > 1040.0 or event['pressure'] < 980.0:
        return (True, 'pressure')

    return (False, None)
```

Listing 1: Função de detecção de anomalias em Python.

Lógica em Apache Spark: No Spark, a mesma lógica é aplicada de forma declarativa sobre um DataFrame. Em vez de um `if/else`, utilizamos a expressão `when().otherwise()` para criar uma nova coluna booleana (`is_anomaly`) que sinaliza os registros anômalos. Esta abordagem permite que o otimizador do Spark integre a detecção diretamente no plano de execução.

```
# Em solution_spark/processor.py
anomaly_conditions = (
    (col("temperature") >= 45.0) | (col("temperature") <= -10.0) |
    (col("humidity") > 100.0) | (col("humidity") < 0.0) |
    (col("pressure") > 1040.0) | (col("pressure") < 980.0)
)

df_with_anomalies = df.withColumn("is_anomaly",
    when(anomaly_conditions, 1).otherwise(0)
)
```

Listing 2: Lógica declarativa de detecção de anomalias no Spark.

3.2 Métrica 2: Média Móvel por Região

Lógica em Python (Multiprocessing e Message Broker): Para calcular a média móvel de forma eficiente, foi utilizada a estrutura de dados `collections.deque` com um tamanho máximo definido (`maxlen`). Um `deque` funciona como uma janela deslizante: ao adicionar um novo elemento quando a capacidade máxima é atingida, o elemento mais antigo é automaticamente descartado. Apenas eventos não anômalos são adicionados ao `deque`.

```
# Em core/metrics.py
temp_window = deque(maxlen=window_size)
hum_window = deque(maxlen=window_size)
press_window = deque(maxlen=window_size)

for event in events:
    anomaly_found, _ = is_anomalous(event)
    if not anomaly_found:
        temp_window.append(event['temperature'])
        # ... (logica similar para umidade e pressao) ...
```

Listing 3: Uso de `deque` para média móvel em Python.

Lógica em Apache Spark: A média móvel no Spark é implementada de forma idiomática através de **Funções de Janela (Window Functions)**. Uma especificação de janela é definida para particionar os dados por região (`partitionBy`), ordená-los por tempo (`orderBy`), e definir os limites da janela como as 50 linhas anteriores à linha atual (`rowsBetween`). A função de agregação `avg()` é então aplicada sobre esta janela.

```
# Em solution_spark/processor.py
window_region = Window.partitionBy("region") \
                      .orderBy("timestamp") \
                      .rowsBetween(-49, 0) # 49 linhas antes + a atual =
    50

# ...
region_moving_avg_report = df_no_anomalies.withColumn(
```

```
    "temp_mov_avg", avg("temperature").over(window_region)
).groupBy("region").agg(...)
```

Listing 4: Definição e uso de Window Function para média móvel no Spark.

3.3 Métrica 3: Períodos de Anomalias Múltiplas

Lógica em Python (Multiprocessing e Message Broker): Similarmente à média móvel, um `deque` é usado como janela deslizante. No entanto, em vez de um tamanho fixo, a janela é baseada em tempo. Para cada evento, a função primeiro remove do início do `deque` todos os eventos que são mais antigos que 10 minutos em relação ao evento atual. Em seguida, verifica se os eventos restantes na janela contêm anomalias de mais de um tipo de sensor.

```
# Em core/metrics.py
window = deque()
for event in events:
    event_time = datetime.fromisoformat(event['timestamp'])

    # Remove eventos antigos da janela
    while window and (event_time - window[0]['timestamp_obj'] >
           timedelta(minutes=window_minutes)):
        window.popleft()
    # ... (adiciona anomalia atual e verifica sensores) ...
```

Listing 5: Janela temporal com `deque` em Python.

Lógica em Apache Spark: Esta métrica também é resolvida com Funções de Janela, mas de uma forma mais avançada. Em vez de uma janela baseada em contagem de linhas (`rowsBetween`), utiliza-se uma janela baseada em intervalo de valores (`rangeBetween`). A coluna de ordenação é o timestamp convertido para segundos (Unix time), e o intervalo é definido como 600 segundos (10 minutos). Isso garante que a janela sempre represente o período de tempo correto, independentemente da frequência dos eventos, sendo uma abordagem mais robusta.

```
# Em solution_spark/processor.py
window_station_10min = Window \
    .partitionBy("station_id") \
    .orderBy(unix_timestamp("timestamp")) \
    .rangeBetween(-600, 0) # Janela de 600 segundos (10 min)

multi_anomaly_periods = df_with_anomalies.filter(...).withColumn(
    "distinct_anomaly_sensors_in_window",
    count(col("anomaly_sensor")).over(window_station_10min)
)
```

Listing 6: Janela temporal com `rangeBetween` no Spark.

4 Modelagem das Arquiteturas de Solução

A seguir, detalha-se a arquitetura de cada uma das três soluções implementadas.

4.1 Solução 1: Multiprocessamento Local

A primeira abordagem implementada visa explorar o paralelismo em uma única máquina, utilizando os múltiplos núcleos de CPU disponíveis. Para isso, foi utilizado o módulo `multiprocessing` do Python. A estratégia central desta solução foi projetada para ser eficiente em termos de memória no processo principal, evitando carregar todo o conjunto de dados de uma só vez. O processamento é dividido em três etapas principais: divisão do trabalho, processamento paralelo e agregação dos resultados.

4.1.1 Etapa 1: Divisão do Trabalho por Chunks de Arquivo

Em vez de carregar todo o arquivo CSV na memória, o processo principal primeiro divide o arquivo em "pedaços"(chunks) de bytes aproximadamente iguais. Esta tarefa é realizada pela função `get_file_chunks` no módulo `data_parser.py`. A função calcula os pontos de início e fim de cada chunk, com o cuidado de avançar até a próxima quebra de linha para evitar a divisão de uma mesma linha de dados entre dois workers.

```
def get_file_chunks(data_path: str, num_chunks: int):
    # ... (calculo do tamanho do chunk) ...
    with open(data_path, 'rb') as f:
        # ... (pula o cabecalho) ...
        for i in range(num_chunks):
            # ...
            f.seek(start_byte + chunk_size)
            # Avanca ate o fim da linha para nao corta-la
            f.readline()
            end_byte = f.tell()
            chunks.append((start_byte, end_byte))
            start_byte = end_byte
            # ...
    return chunks
```

Listing 7: Lógica principal da função `get_file_chunks`.

O resultado é uma lista de tuplas, onde cada tupla (start_byte, end_byte) define a fatia do arquivo que um processo worker será responsável por ler e processar.

4.1.2 Etapa 2: Processamento Paralelo por Worker

A lista de chunks é então distribuída para um `multiprocessing.Pool`, um pool de processos workers. Cada worker executa a função `process_file_chunk`, que realiza as seguintes ações:

1. Recebe os bytes de início e fim que lhe foram designados.
2. Abre o arquivo de dados original e busca (`f.seek()`) diretamente a sua posição inicial.
3. Lê seu chunk de bytes para um buffer em memória (`io.StringIO`), tratando-o como um arquivo CSV independente.
4. Itera sobre as linhas do seu chunk, convertendo os tipos de dados e aplicando a lógica de negócio, como a detecção de anomalias e o cálculo das métricas.

```
def process_file_chunk(args: tuple[str, int, int]):
    data_path, start_byte, end_byte = args

    with open(data_path, 'r', newline='') as f:
        f.seek(start_byte)
        chunk_size = end_byte - start_byte
        chunk_buffer = io.StringIO(f.read(chunk_size))

    reader = csv.reader(chunk_buffer)
    # ... (resto do processamento) ...
```

Listing 8: Worker lendo seu chunk de dados designado.

Ao final de seu trabalho, cada worker retorna um dicionário contendo os resultados parciais que calculou, incluindo a lista de anomalias que encontrou em seu chunk específico.

4.1.3 Etapa 3: Agregação dos Resultados

O processo principal, que estava bloqueado aguardando a finalização de todos os workers do pool, recebe uma lista de dicionários com os resultados parciais. A etapa final consiste em uma agregação rápida e sequencial desses resultados para construir os relatórios finais e a lista completa de anomalias detectadas.

```
def run_analysis(data_path: str, num_workers: int):
    # ... (cria o pool e executa pool.map) ...

    all_found_anomalies = []
    for res in partial_results:
        # ... (agrega metricas) ...
        if res.get("found_anomalies"):
            all_found_anomalies.extend(res['found_anomalies'])

    return (end_time - start_time), all_found_anomalies
```

Listing 9: Agregação final dos resultados no processo principal.

4.2 Solução 2: Message Broker com Workers

A segunda abordagem implementa um padrão de processamento distribuído utilizando um *message broker* (RabbitMQ) para desacoplar a produção de tarefas do seu consumo. Para maximizar o desempenho e evitar os gargalos de um padrão Map-Reduce tradicional, foi adotada a arquitetura de **Workers Agregadores**. Nesta estratégia, a carga de agregação de dados é distribuída entre os próprios workers, em vez de ser centralizada em um único processo redutor.

O fluxo de trabalho completo é dividido em três etapas distintas.

4.2.1 Etapa 1: Produção e Enfileiramento de Tarefas

O processo é iniciado pelo `producer.py`, cuja única responsabilidade é ler o arquivo de dados e transformar cada linha em uma tarefa independente. Cada linha do CSV é serializada para o formato JSON e publicada como uma mensagem persistente na fila `task_queue` do RabbitMQ.

```
# Em solution_message_broker/producer.py
def run_producer(data_path: str):
```

```
    # ... (conex o com RabbitMQ) ...
    for row in reader:
        message_body = json.dumps(row)
        channel.basic_publish(
            exchange='',
            routing_key='task_queue',
            body=message_body,
            properties=pika.BasicProperties(
                delivery_mode=pika.spec.PERSISTENT_DELIVERY_MODE
            )
        )
```

Listing 10: Publicação de uma mensagem na fila pelo Producer.

Essa abordagem garante que todas as tarefas estejam disponíveis no broker antes mesmo de os workers começarem a processá-las.

4.2.2 Etapa 2: Processamento e Agregação Paralela nos Workers

Esta é a etapa central da otimização. Múltiplos processos `worker.py` são iniciados em paralelo pelo orquestrador. Cada worker é independente e executa o seguinte ciclo de vida:

1. **Consumo em Lote:** O worker se conecta à fila `task_queue` e consome o máximo de mensagens que conseguir. Um `inactivity_timeout` é utilizado para que o worker pare de consumir e considere sua "porção"de trabalho concluída quando a fila fica vazia por um breve período.

```
# Em solution_message_broker/worker.py
station_events = defaultdict(list)
for method_frame, properties, body in channel.consume(
    'task_queue', inactivity_timeout=5
):
    if method_frame is None: break
    event = json.loads(body)
    # ... (convers o de tipos e armazenamento em mem ria) ...
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)

```

Listing 11: Loop de consumo de mensagens no Worker.

2. **Agregação Local:** Após sair do loop de consumo, cada worker possui um subconjunto dos dados totais em sua própria memória. Ele então executa **todas as lógicas de cálculo de métricas e detecção de anomalias** sobre este subconjunto local, agindo como um "mini-redutor".

3. **Publicação do Resultado Agregado:** Ao final, em vez de retornar milhões de eventos processados, o worker calcula um resultado final e publica **uma única mensagem** contendo um sumário de suas descobertas (métricas e lista de anomalias) na `result_queue`.

```
# Em solution_message_broker/worker.py
final_result = {
    "station_metrics": worker_station_report,
    "found_anomalies": worker_found_anomalies
}
```

```
channel.basic_publish(
    exchange='',
    routing_key='result_queue',
    body=json.dumps(final_result)
)
```

Listing 12: Publicação do resultado único e agregado pelo Worker.

4.2.3 Etapa 3: Consolidação Final dos Resultados

O processo orquestrador, `processor.py`, após iniciar os workers e aguardar a sua finalização, realiza a última etapa. Ele se conecta à `result_queue` para consumir os poucos resultados agregados (um para cada worker que processou dados). A agregação final é leve e rápida, consistindo basicamente em concatenar as listas de anomalias encontradas por cada worker.

```
# Em solution_message_broker/processor.py
all_found_anomalies = []
for method_frame, properties, body in channel.consume(
    'result_queue', inactivity_timeout=5
):
    if method_frame is None: break
    result = json.loads(body)
    if result.get("found_anomalies"):
        all_found_anomalies.extend(result["found_anomalies"])
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)
```

Listing 13: Coleta dos resultados agregados no Processador.

Vale ressaltar que no contexto de um experimento em máquina única, esta solução demonstra o mais alto **overhead de comunicação**. Cada evento é serializado para JSON, enviado através da rede (simulada pelo Docker) para o broker, e depois desserializado pelo worker. Esse processo, repetido para milhares de eventos, é significativamente mais lento do que o acesso direto à memória realizado pelas outras duas abordagens. Portanto, o propósito de incluir esta solução no experimento é quantificar o custo de desempenho de uma arquitetura distribuída e desacoplada, que é pago em troca de escalabilidade e resiliência.

4.3 Solução 3: Apache Spark

A terceira abordagem utiliza o Apache Spark, um motor de processamento de dados distribuído de alto desempenho, para executar a análise. Diferente das abordagens anteriores, que são imperativas (descrevem "como"fazer), a solução com Spark utiliza a API declarativa de DataFrames. Isso permite que o desenvolvedor especifique "o que"quer fazer, enquanto o motor do Spark, através de seu otimizador Catalyst, determina o plano de execução mais eficiente.

Para este experimento, o Spark foi configurado para rodar em modo `local[*]`, o que instrui o sistema a utilizar todos os núcleos de CPU disponíveis na máquina local, simulando um ambiente de processamento paralelo.

4.3.1 Etapa 1: Inicialização e Ingestão de Dados

O ponto de entrada de qualquer aplicação Spark é a `SparkSession`. Ela é configurada dinamicamente para utilizar o número de workers solicitado pelo experimento.

```
spark = SparkSession.builder \
    .appName(f"Analysis_Workers_{num_workers}") \
    .master(f"local[{num_workers}]") \
    .config("spark.sql.session.timeZone", "UTC") \
    .getOrCreate()
```

Listing 14: Criação da SparkSession.

Os dados são lidos do arquivo CSV diretamente para um DataFrame. Em vez de inferir o esquema (o que pode ser custoso e impreciso), os tipos de dados de cada coluna são explicitamente convertidos (`cast`), garantindo a robustez e a previsibilidade do processamento.

4.3.2 Etapa 2: Transformações, Lazy Evaluation e Otimização

No Spark, a maioria das operações, como filtros e adições de colunas, são *transformações*. Elas não são executadas imediatamente. Em vez disso, o Spark constrói um plano de execução lógico na forma de um Grafo Acíclico Dirigido (DAG). Esta característica, conhecida como **Lazy Evaluation**, permite ao otimizador Catalyst analisar o grafo completo e encontrar a forma mais eficiente de executar o trabalho, aplicando otimizações como a reordenação de operações.

Para identificar as anomalias, uma nova coluna é adicionada ao DataFrame usando uma expressão `when`, uma forma declarativa e poderosa de aplicar lógica condicional sobre os dados.

```
anomaly_conditions = (
    (col("temperature") < 5.0) | (col("temperature") > 45.0) |
    # ... outras condicoes ...
)

df_with_anomalies = df.withColumn(
    "is_anomaly", when(anomaly_conditions, 1).otherwise(0)
)
```

Listing 15: Identificação de anomalias de forma declarativa.

Como o DataFrame `df_with_anomalies` é a base para múltiplos cálculos subsequentes, utilizamos o método `.cache()`. Isso instrui o Spark a manter este DataFrame intermediário na memória após sua primeira computação, evitando que ele seja recalculado do zero para cada métrica, uma otimização de desempenho crucial.

4.3.3 Etapa 3: Cálculo de Métricas com Agregações e Funções de Janela

Com o DataFrame base preparado e em cache, as métricas são calculadas usando as operações de alto nível do Spark.

Agregações Simples (groupBy): Para métricas como o percentual de anomalias por estação, a operação `groupBy("station_id").agg(...)` é utilizada para agrupar todos os dados de uma mesma estação e aplicar funções de agregação, como a contagem de eventos.

Funções de Janela (Window Functions): Para cálculos mais complexos que dependem de um conjunto de linhas relacionadas à linha atual, as Funções de Janela são a ferramenta ideal.

- **Média Móvel:** Foi definida uma janela baseada na contagem de linhas (`rowsBetween`) para calcular a média dos últimos 50 eventos não anômalos de uma mesma região.

```
window_region = Window.partitionBy("region") \
                      .orderBy("timestamp") \
                      .rowsBetween(-49, 0)

df.withColumn("temp_mov_avg", avg("temperature").over(window_region))

```

Listing 16: Definição da janela para a média móvel.

- **Anomalias Múltiplas:** Para esta métrica, foi utilizada uma janela baseada em intervalo de tempo (`rangeBetween`).

```
window_station_10min = Window.partitionBy("station_id") \
                             .orderBy(unix_timestamp("timestamp")) \
                             .rangeBetween(-600, 0)

df.withColumn("sensors_in_window", count(...).over(window_station_10min))

```

Listing 17: Definição da janela de tempo para anomalias múltiplas.

4.3.4 Etapa 4: Execução e Coleta de Resultados

Nenhuma das operações acima é executada até que uma *ação* seja chamada. O método `.collect()` é uma ação que dispara a execução de todo o plano otimizado pelo Spark. Ele computa os resultados de forma distribuída e os retorna para o processo principal (o "driver"Python). Ao final, a sessão Spark é encerrada com `spark.stop()` para liberar os recursos.

O Spark oferece a solução mais robusta, escalável e performática para cargas de trabalho de dados, especialmente em volumes maiores e com maior paralelismo.

5 Resultados Experimentais e Dashboard

Nesta seção, apresentamos os dados quantitativos coletados através do dashboard do experimento.

5.1 Resultados de Desempenho

As imagens abaixo apresentam o tempo de execução (em segundos) de cada abordagem para os diferentes graus de paralelismo testados.

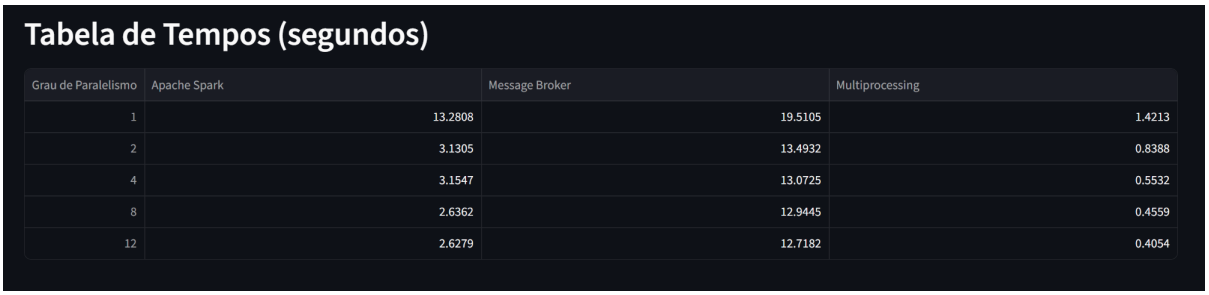

Figura 1: Tabela Tempo x Abordagem x Grau de Paralelismo

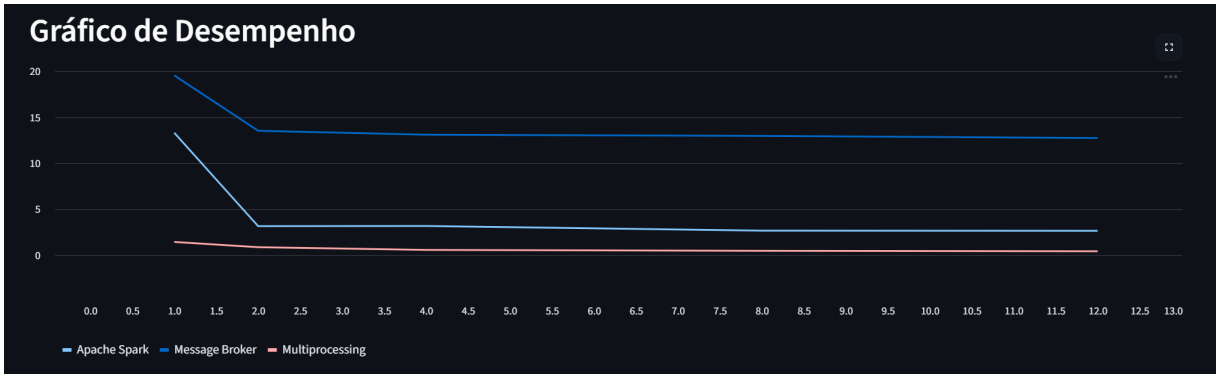

Figura 2: Gráfico de Desempenho: Tempo vs. Grau de Paralelismo

6 Discussão e Análise dos Resultados

A execução do experimento em diferentes cenários de paralelismo forneceu resultados quantitativos claros, permitindo uma análise aprofundada das características de desempenho de cada uma das três arquiteturas propostas. Os dados de tempo de execução, apresentados na Tabela 1 e no Gráfico 1, confirmam algumas previsões teóricas, mas também revelam nuances importantes sobre a performance em um ambiente de máquina única com um volume de dados moderado.

Tabela 1: Tabela de Tempos (segundos) vs. Grau de Paralelismo.

Paralelismo	**Multiprocessing**	**Apache Spark**	**Message Broker**
1	1.4213	13.2808	19.5105
2	0.8388	3.1305	13.4932
4	0.5532	3.1547	13.0725
8	0.4559	2.6362	12.9445
12	0.4054	2.6279	12.7182

6.1 Análise do Multiprocessing

Conforme os dados, a abordagem com Multiprocessing foi a mais performática em todos os cenários testados, iniciando em 1.42s com um único processo e escalando de forma

excelente até atingir apenas 0.40s com 12 processos.

Este resultado é explicado pelo overhead de comunicação praticamente nulo desta arquitetura. Os dados são lidos do disco e processados na memória da mesma máquina, e a comunicação entre o processo principal e os workers é gerenciada pelo sistema operacional de forma extremamente eficiente (via mecanismos de IPC). Não há custos de serialização de dados para formatos como JSON, nem latência de rede, fatores que impactam as outras duas soluções.

A curva de desempenho para o Multiprocessing (a linha mais baixa no Gráfico começa a convergir com 8 núcleos, vemos uma diferença de menos de 0,05 segundos de 8 para 12. Isso demonstra sua principal limitação: a escalabilidade vertical. O ganho de desempenho é limitado pelo número de núcleos físicos da máquina. Embora tenha sido a solução vencedora neste experimento, ela não seria capaz de escalar para um cluster de múltiplas máquinas para processar volumes de dados na ordem de terabytes. Essa abordagem é ideal para tarefas de processamento pesado ("CPU-bound") em uma única máquina.

6.2 Análise do Apache Spark

A solução com Apache Spark apresentou o comportamento mais interessante. Começou com um tempo de execução elevado (13.28s com 1 core), mas demonstrou a melhor capacidade de escalabilidade relativa, reduzindo seu tempo em mais de 75% ao passar para 2 cores. A partir daí, continuou a melhorar, terminando em 2.62s com 12 cores.

O alto tempo inicial é um reflexo direto do overhead de inicialização da Java Virtual Machine (JVM) e da criação da SparkSession. Este é um custo fixo que o Spark paga em cada execução. Contudo, uma vez que o ambiente está no ar, o motor de execução do Spark processa os dados em memória de forma colunar e otimiza o plano de execução, o que explica o ganho ao adicionar mais núcleos.

Se o volume de dados fosse 100 vezes maior, é quase certo que o tempo de processamento do Spark superaria o da solução de Multiprocessing. O trade-off aqui é: paga-se um custo inicial de complexidade e inicialização em troca de um poder de processamento e escalabilidade que são ordens de magnitude maiores, sendo a escolha padrão para ecossistemas de Big Data.

6.3 Análise do Message Broker

Como previsto teoricamente, a arquitetura com Message Broker foi a mais lenta em todos os testes, começando em 19.51s e escalando de forma pouco eficiente para 12.71s.

O desempenho inferior é consequência direta do alto overhead de serialização e comunicação. Cada um dos eventos no arquivo de dados passou pelo seguinte ciclo: (1) lido do CSV, (2) serializado para JSON, (3) enviado pela rede do Docker ao RabbitMQ, (4) consumido pelo worker, (5) desserializado de JSON para um objeto Python para então ser processado. Este ciclo, repetido milhares de vezes, impõe uma latência significativa que não existe nas outras abordagens.

É fundamental ressaltar que o objetivo desta arquitetura não é a performance máxima em uma única máquina. Seus benefícios — não medidos neste experimento — são o desacoplamento, a resiliência e a escalabilidade horizontal. Em um sistema de produção, onde workers podem rodar em dezenas de máquinas diferentes e tarefas podem ser reenfileiradas em caso de falha, esta arquitetura oferece características que as outras não

possuem. O experimento, portanto, serviu para quantificar o "preço"em performance que se paga para obter esses benefícios arquitetônicos.

6.4 Conclusão

A análise comparativa das três soluções de paralelismo revelou um claro trade-off entre overhead, desempenho e potencial de escalabilidade. Os resultados mostraram que para o caso de um grande um volume de dados (falando na casa de milhões, maior do que fomos capazes de processar nesse experimento) o processamento com Spark passa a ser a melhor opção, logo esta seria a abordagem recomendada para a rede ClimaData para calcular apenas algumas métricas, porém provenientes de uma quantidade massiva de dados.