Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

PAULO CESAR ROMANO

PARADIGMA FUNCIONAL EM JAVA: UMA ANÁLISE EXPLORATÓRIA ENTRE STREAMS SERIAL E PARALELA

Assis/SP
2022

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

PAULO CESAR ROMANO

PARADIGMA FUNCIONAL EM JAVA: UMA ANÁLISE EXPLORATÓRIA ENTRE STREAMS SERIAL E PARALELA

Projeto de pesquisa apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando(a): Paulo Cesar Romano
Orientador(a): MSc. Guilherme de Cleva Farto

Assis/SP
2022

FICHA CATALOGRÁFICA

R759p Romano, Paulo Cesar.

Paradigma funcional em Java: uma análise exploratória entre Streams serial e paralela / Paulo Cesar Romano – Assis, SP: FEMA, 2022.

73 f.

Trabalho de Conclusão de Curso (Graduação) – Fundação Educacional do Município de Assis – FEMA, curso de Ciência da Computação, Assis, 2022.

Orientador: Prof. M.e Guilherme de Cleva Farto.

1. Java. 2. Programação funcional. 3. API de Stream. I. Título.

CDD 005

Biblioteca da FEMA

PARADIGMA FUNCIONAL EM JAVA: UMA ANÁLISE EXPLORATÓRIA ENTRE STREAMS SERIAL E PARALELA

PAULO CESAR ROMANO

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: _____
Prof. MSc. Guilherme de Cleva Farto

Examinador: _____
Prof. Dr. Almir Rogério Camolesi

Assis/SP
2022

AGRADECIMENTOS

Primeiramente agradeço a Deus por ter me dado a capacidade e perseverança para a conclusão de mais uma etapa.

Agradeço por todo apoio que recebi do meu pai e da minha mãe e de todos os amigos, colegas de trabalho e professores que me incentivaram, auxiliaram e estiveram comigo durante estes anos.

“Moderação na defesa da verdade é serviço prestado à mentira”.

Olavo de Carvalho (1947 – 2022)

RESUMO

Com o advento do Java 8 e a incorporação do paradigma funcional à linguagem em conjunto com as expressões lambdas, uma nova API foi desenvolvida. A API de *Stream* permite manipular coleções de forma simplificada, além de possuir duas vertentes para processamento, sendo estas a serial e a paralela. A escolha de qual modo processar uma coleção pode impactar diretamente o desempenho e performance de uma aplicação, bem como a satisfação do usuário. A proposta deste trabalho consistiu na análise do comportamento desta API em suas duas vertentes, mediante demandas de processamento com complexidade e quantidade de dados variados, a fim de descobrir se haveria diferença entre o modo serial e paralelo e em quais contextos um poderia ter vantagem sobre o outro.

Palavras-chave: Programação funcional; Java; API de *Stream*.

ABSTRACT

With the advent of Java 8, the incorporation of the functional paradigm into the language together with lambdas expressions, a new API was developed. The Stream's API allows handling collections in a simplified way, in addition to having two aspects for processing, which are serial and parallel. Choosing which way to process a collection can directly impact the performance of an application, as well as user satisfaction. The purpose of this work consisted of analyzing the behavior of this API in its two aspects, through processing demands with varying complexity and amount of data, in order to find out if there would be a difference between the serial and parallel mode and in which contexts one could have an advantage over the other.

Keywords: Functional programming; Java; Stream's API.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

O setor de qualidade, um dos pilares na construção de *software,* deve garantir que um programa cumpra o propósito pelo qual ele foi projetado, de forma que, ao ser entregue ao cliente, ele seja funcional e esteja acordo com os requisitos solicitados pelo mesmo (ZANIN; et al, p 11, 2018).

De acordo com Zanin, et al. (p 15, 2018), dentro deste contexto de controle de qualidade de *software*, alguns de seus grandes benefícios estão relacionados com a economia de dinheiro ao não ter que gastar tempo com solução de *bugs*, inconsistência de dados e problemas do sistema. Além disso, o nível de experiência que o usuário tem com o *software* está relacionado também ao seu desempenho. Portanto, sistemas com respostas rápidas resultam em usuários satisfeitos.

Segundo Silva (2019), com o Teste de Performance é possível avaliar um sistema a partir de sua “capacidade de resposta, robustez, disponibilidade, confiabilidade e escalabilidade”, com o intuito de remover problemas relacionados ao seu desempenho.

De acordo com Tucker e Noonan (p 4, 2010), na programação funcional os problemas passam a ser modelados como uma “coleção de funções matemáticas, cada uma com um espaço de entrada (domínio) e resultado (faixa)”. Este arquétipo chegou na linguagem Java a partir da versão 8, feito pela Oracle em 2014. A novidade trouxe consigo alterações em sua sintaxe, além de oferecer mais facilidade aos programadores em tarefas que anteriormente eram mais complexas e exigiam muitas linhas de códigos (SILVA, 2016).

A API de Stream, uma das novidades do Java 8, permite de forma declarativa, trabalhar com operações de agregação a partir da conversão de uma coleção, *array* ou recursos de entrada e saída de dados, gerando, ao final, uma nova coleção ou valor reduzido (AMORIM, 2015).

Segundo Silva (2016), esta API também torna possível trabalhar com o paradigma funcional de forma paralela, ou seja, ela é capaz de dividir uma tarefa em subtarefas, processá-las paralelamente e combinar, por último, o resultado final.

Embora por um lado a forma paralela de se trabalhar com a API de Stream possa melhorar o desempenho e aproveitar de modo mais abrangente o potencial do processador, não

obstante, em certos casos o custo presente nesta abordagem envolvendo tarefas adicionais para o processamento pode torná-la inviável, não compensando a sua utilização.

1.1. OBJETIVOS

Este trabalho tem como principal objetivo, de forma exploratória, analisar aspectos e cenários de tempo de processamento e uso do sistema operacional, por meio do paradigma funcional serial e paralelo em Java aplicado a diversos contextos para avaliar a abordagem de programação funcional na plataforma Java e seus recursos nativos.

1.2. JUSTIFICATIVA

Dentro do contexto de planejamento de software, um dos requisitos de grande importância que deve ser levado em consideração é o tempo que o sistema leva para processar uma determinada tarefa, ou seja, a sua performance. Este fator em certos sistemas que exigem respostas rápidas pode até mesmo gerar problemas financeiros e riscos à saúde humana. Portanto, de modo geral, a performance de um sistema tem impacto direto com a satisfação ou não do usuário (AMORIM, 2014).

Segundo Silva (2019), uma pesquisa feita em conjunto entre a Akamai e Forrester Consulting chegou à conclusão de que “40% dos consumidores não esperam mais do que 3 segundos pelo carregamento de uma página”. Para mais, dos usuários que estejam com um carrinho de compras, 51% deles acabam abandonando devido à demora do sistema em processar a solicitação.

Dito isto, a justificativa para elaboração deste trabalho permeia a utilização da API de *Stream* serial e paralela do Java, como alternativa para trabalhar com coleções de forma mais eficiente, com um custo menor de tempo e maior desempenho ao sistema.

1.3. MOTIVAÇÃO

O presente projeto de pesquisa tem por motivação investigar as características e o comportamento entre as abordagens serial e paralela da API de *Stream* por meio de implementações contendo um mesmo fluxo de processamento, com a mesma quantidade de dados, a fim de descobrir quais resultados serão produzidos.

Desta maneira, a partir dos dados produzidos, espera-se contribuir com informações sobre desempenho em relação ao tempo, uso do sistema operacional e as possíveis vantagens e desvantagens da utilização de cada abordagem.

1.4. PERSPECTIVAS DE CONTRIBUIÇÃO

A chegada da programação funcional e as demais *features* oriundas do Java 8 trouxeram mais flexibilidade, legibilidade e limpeza no código, deixando a cargo do desenvolvedor somente “o que” ele deve fazer, além de facilitar o uso da programação paralela por meio do paradigma funcional. O presente trabalho pretende apresentar informações de possível interesse aos desenvolvedores Java que desejam ingressar no paradigma funcional ou se aprofundarem nele, de tal forma que ao final eles possam compreender os conceitos e as características que constituem este paradigma dentro da linguagem, bem como dar um embasamento sólido para qual abordagem, dentro do seu contexto de desenvolvimento, será mais benéfica e performática.

1.5. METODOLOGIA

A metodologia de pesquisa definida para o presente trabalho consistirá inicialmente no levantamento bibliográfico, onde será realizado um estudo exploratório com o propósito de fundamentar os conceitos que estarão presentes posteriormente nas implementações.

Os conceitos a serem explanados envolvem o paradigma funcional e sua aplicação dentro da linguagem Java mediante as Interfaces Funcionais; as características e funcionalidades presentes na API de *Stream*, tanto serial como paralela e o encadeamento de operações (*pipeline*) que esta API proporciona.

Após a finalização de todo o embasamento teórico, será dado prosseguimento as implementações do projeto, onde serão realizados os testes. Estes testes serão executados a partir de uma mesma base de dados. Para o processamento destes dados, será utilizado a programação funcional em Java com diversas operações disponibilizadas pela API de *Stream*. Estas mesmas operações serão executadas tanto na *stream* serial quanto na paralela. Ao encerrar o ciclo dos testes será aumentado o volume de dados com a intenção de refazer as mesmas operações processadas anteriormente e obter novos resultados.

Concluída esta etapa, os dados adquiridos pelos testes serão agrupados em gráficos com base no tempo de processamento, operações realizadas e quantidade de dados utilizada. Por último, os resultados obtidos serão analisados de forma quantitativa, para finalmente chegar à conclusão sobre a performance, possíveis vantagens e desvantagens que as abordagens serial e paralela poderão trazer.

2. LINGUAGENS DE PROGRAMAÇÃO

As linguagens de programação, como forma de facilitar a comunicação e expressões de ideias, assim como as linguagens naturais, possibilitam a interação entre pessoas e computadores através de ideias computacionais. Esta comunicação, no entanto, é mais limitada devido ao domínio de expressão que as linguagens possuem (TUCKER; NOONAN, 2010, p 1).

Segundo Tucker e Noonan (p 2-3, 2010), o vocabulário básico para os projetistas de linguagens, conhecido como “princípios de projeto de linguagens” é dividido em três principais categorias que englobam a Sintaxe, ou seja, o que a linguagem define como um programa estruturado de forma correta, bem como a sua gramática, palavras, símbolos e formalismos linguísticos; os Nomes e Tipos que dizem respeito ao vocabulário da linguagem com suas regras de nomenclatura de entidades, além dos tipos de valores que são possíveis de serem manipulados por programas, desde os mais simples, como valores numéricos e booleanos, os estruturados, como as *strings* e listas, até os que possuem maior complexidade, como é o caso das funções e classes; e por fim a Semântica, que define o significado de um programa, isto é, como os resultados obtidos com a utilização de cada comando refletem em sua execução.

Dito isso, os recursos disponibilizados por uma linguagem e a forma como estes serão utilizados para o desenvolvimento de programas, levando em consideração o fator qualidade, dependem de certas características que precisam ser levadas em consideração ao projetar uma linguagem de programação, tais como os tipos de problemas que a linguagem se destina a solucionar (requisitos) e a forma de apresentar a resolução destes problemas da maneira mais natural (expressividade); o paradigma que melhor se adequa ao contexto dos problemas envolvidos, a possibilidade ou não de implementar a solução proposta e se a solução destes requisitos são eficientes (MELO; SILVA, 2014, p 8).

2.1. PARADIGMAS

De acordo com Tucker e Noonan (p 3, 2014), paradigmas representam uma forma padronizada em que os problemas são estruturados e representados em uma determinada linguagem de programação. Algumas destas linguagens, podem, inclusive, suportar mais de um paradigma.

> Os diferentes paradigmas também podem envolver o domínio da aplicação das linguagens de programação, tais como: desenvolvimento de sistemas para uso comercial (Sistemas de Informação), aplicações matemáticas e aprendizado de programação, entre outros (SILVEIRA; et al, 2021, p 7).

Como aponta Silva, Leite e Oliveira (p 14, 2019), ao longo das décadas os paradigmas vieram evoluindo, e estes são classificados em quatro tipos: paradigma imperativo, funcional, lógico e orientado a objetos.

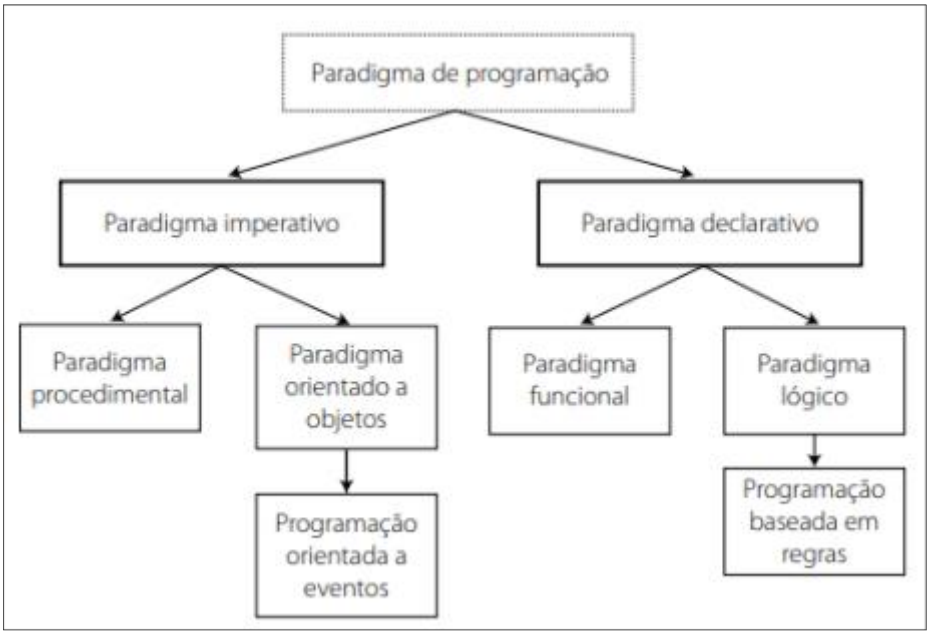


Figura 1: Hierarquia dos paradigmas de programação (In: SILVA; LEITE; OLIVEIRA, p 14, 2019)

2.1.1. PARADIGMA DECLARATIVO

Surgido durante a década de 1970, o paradigma declarativo se colocou como uma nova vertente na área da programação, visto que o programador se preocuparia no “quê” ele deveria fazer para construir as novas funcionalidades durante o desenvolvimento do programa, ao invés de ter que descrever passo a passo de “como” fazê-las para obter o mesmo resultado, como é o caso do paradigma imperativo (SILVA; LEITE; OLIVEIRA, 2019, p 25).

Para Silva, Leite e Oliveira (p 26, 2019), portanto, a base fundamental deste modelo de programação está centrada sempre na declaração, seja por meio de condicionais, comportamentos ou outras formas que delinearão o fluxo de um programa.

2.1.2. PARADIGMA FUNCIONAL

Surgido na década de 1960, o aparecimento da programação funcional se deu graças a falta de aplicabilidade e por não conseguir satisfazer as necessidades dos pesquisadores da época com relação ao desenvolvimento da inteligência artificial e suas ramificações. A causa, no entanto, destas debilidades, se davam pelo fato de as linguagens serem baseadas no paradigma imperativo (TUCKER; NOONAN, 2014, p 361).

Para Silva, Leite e Oliveira (p 122, 2019), o paradigma funcional é constituído por funções matemáticas e, portanto, possibilita estruturar programas de tal modo que, problemas complexos que necessitem ser processados, sejam simplificados até assumirem uma forma normal.

Além disso, ao contrário das linguagens imperativas em que o mapeamento ordenado de expressões é feito de forma sequencial e por iteração, o paradigma funcional por ser baseado em funções matemáticas, gerencia este mapeamento através de recursão e expressões condicionais (SEBESTA, 2018, p 634).

Segundo Pereira (2015), o novo arquétipo utilizado para a construção do fluxo de algoritmo pela programação funcional se deve ao conceito introduzido pelo paradigma declarativo que, além de melhorar a legibilidade do código, também resguarda o mesmo de possíveis

efeitos colaterais (*side-effects*) e preserva a variável de mutabilidades, gerando, portanto, códigos mais concisos.

Portanto, a implementação de códigos por meio deste paradigma possibilita a visualização uniforme das funções que constituem os programas, possui notação concisa, traz mais facilidade na realização de testes e na busca por *bugs*¸ gerencia o uso de memória de modo automático, trata as funções como dados, possui grande flexibilidade e semântica simples (KEOMA, 2018).

- **Cálculo Lambda:** Segundo Costa (2015), Alonzo Church foi quem fundamentou a teoria do Cálculo Lambda que define “um sistema formal para definições, aplicação e recursão de funções”.

  Este cálculo, desenvolvido por Church é uma forma matemática de representar a computação através de funções anônimas – sendo elas tipadas ou não – e da geração de operadores mais complexos a partir da combinação de operadores e funções básicas (REZENDE, 2018).

  Para Silva, Leite e Oliveira (p 137, 2019), considerado uma linguagem intermediária, o cálculo lambda “permite implementar de forma mais eficaz as funções matemáticas a nível de linguagem de máquina”.

  Por meio deste cálculo, ocorre a substituição parcial da expressão que compõe a função. Esta substituição deve seguir três exigências primordiais para a reescrita do programa que será reduzido (MELO; SILVA, 2014, p 130):

  1. **Correção:** cada uma das regras, individualmente, deve preservar o significado das expressões. Ou seja, a interpretação de uma expressão antes e depois de uma redução deve ser a mesma.
  2. **Confluência:** se mais de uma regra for aplicável a uma expressão, a ordem de aplicação deve ser indiferente.
  3. **Terminação:** o conjunto de regras não deve permitir, para qualquer expressão, que uma sequência de reduções produza uma expressão idêntica a ela, caso contrário tal expressão não poderia ser calculada.

  A Correção, como uma propriedade semântica, a fim de validar expressões, exige uma formalidade para a codificação de funções e observância dos parâmetros existentes. Além disso, a Confluência e Terminação se caracterizam como

propriedades sintáticas, pois elas não dependem da interpretação dos símbolos feita pelo programa. Como resultado desta combinação, as estratégias adotadas para redução de expressões as tornam eficientes e garantem a resolução de problemas codificados como um programa puramente funcional (MELO; SILVA, p 131, 2014).

- **Expressões Lambda:** Também conhecida como “funções lambda”, as expressões lambda são constituídas por blocos de códigos, que podem ser utilizadas em diversos contextos, tais como a atribuição desta função a uma variável e a aplicação da função como argumento ou retorno de outras funções (ATENCIO, 2015).
  Para mais, estas funções por definição são anônimas e, por conseguinte, não pertencem a nenhuma classe. Dito isso, estas expressões são normalmente aplicadas em operações que envolvem a filtragem e extração de dados de uma coleção, bem como o percurso entre os seus elementos (SILVA; LEITE; OLIVEIRA, 2019, p 126).

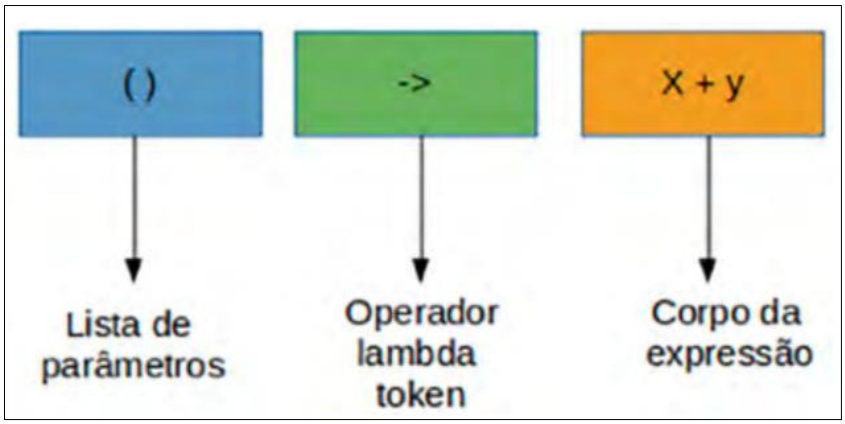


Figura 2: Sintaxe de uma expressão lambda (In: SILVA; LEITE; OLIVEIRA, p 126, 2019)

A estrutura dessa expressão é formada pelos parâmetros de entrada da função – caso haja algum – que antecedem a seta (->) e o que vem posteriormente a ela, isto é, o corpo da expressão com o comportamento que será executado (SILVA; LEITE; OLIVEIRA, 2019, p 126).

- **Avaliação de Função:** Há neste paradigma uma característica importante no que diz respeito a forma em que as linguagens avaliam uma função. Separada em duas vertentes, a avaliação de uma função pode ser classificada como “avaliação rápida”

ou “chamada por valor”. Neste arquétipo, os parâmetros passados para a função são avaliados apenas uma vez no momento de sua chamada, o que, consequentemente, resulta em uma maior eficiência. Por outro lado, a “avaliação lenta”, como estratégia alternativa, trabalha com a avaliação dos argumentos de forma adiada, ou seja, a avaliação ocorre somente quando o argumento passa a ser necessário para a função. Porém, neste modelo, um mesmo argumento passa por uma reavaliação a cada vez que ele é utilizado. Apesar disso, a avaliação lenta possui a vantagem de implementar certas funções que não são possíveis de serem implementadas na avaliação rápida (TUCKER; NOONAN, 2010, p 365).

- **Funções Puras e Impuras:** As funções puras e impuras são classificadas baseadas na imutabilidade da variável em uma expressão. As funções puras, respeitando a noção matemática, “eliminam a noção de célula de memória de uma variável”, por consequência, estas funções não utilizam o operador de atribuição. Porém, se a função reter, em algum momento, algum operador de atribuição, ela é denominada como impura (TUCKER; NOONAN, 2010, p 363).

- **Transparência Referencial:** Esta particularidade está presente em casos em que uma função “depende apenas dos valores de seus argumentos” (SILVA; LEITE; OLIVEIRA, p 136, 2019) e, portanto, isto a torna independente do contexto externo e a caracteriza como determinística, pois dado um mesmo valor de entrada, a função sempre gerará o mesmo resultado na saída (SAUMONT, 2017, p 6).

- **Imutabilidade:** A imutabilidade, como uma das características deste paradigma, consiste na inalteração do valor – uma vez que este é atribuído a função –, até o final do processo. Graças a isso, torna-se mais simples a realização de testes e a solução de *bugs* por não haver alterações no estado da variável. No entanto, por conta desta imutabilidade, há um consumo mais elevado de memória, pois se trabalha com cópias ao invés de referências (SILVA; LEITE; OLIVEIRA, 2019, p 136).

- **Funções de Ordem Superior:** As funções de ordem superior, também conhecidas como “forma funcional” são aquelas que permitem receber uma ou mais funções como argumentos e/ou retornam uma função como resultado. Por meio das funções de ordem superior que recebem duas funções por parâmetro também é possível realizar o que é denominado de “composição funcional”, onde o resultado de uma das funções é passado como argumento da outra função (SEBESTA, 2018, p 635).

- **Tipagem:** Além das linguagens que observam a formulação primária do cálculo Lambda, onde as variáveis são flexíveis para assumirem diferentes tipos ao valor que é atribuído a elas, há, em contrapartida, linguagens que usufruem o paradigma funcional com uma tipagem forte, isto é, as variáveis dispõem de tipos definidos, onde estes tipos passam por uma verificação para validar o valor que será atribuído a elas (MELO; SILVA, 2014, p 134).

3. *THREADS*, PARALELISMO E *PIPELINING*

Para Medeiros (2007), os programas ou processos normalmente dispõem de um fluxo de controle para que sejam executados de forma sequencial. No entanto, dentro deste contexto, torna-se possível a existência de mais de um fluxo de controle graças ao mecanismo de *threads*, pois estas concedem aos programas executarem partes do seu código paralelamente.

Outrossim, uma *thread*, como uma sub-rotina, permite ao programa executá-la de modo assíncrono ao seu respectivo programa. Portanto, “um ambiente *multithread* possibilita a execução concorrente de sub-rotinas dentro de um mesmo processo” (MACHADO; MAIA, p 84, 2017). Além do mais, estas *threads*, pertencentes a um processo, por serem executadas no modo usuário, são conhecidas como “*threads* de usuário” (TEDESCO, 2020).

A execução de tarefas por intermédio do paralelismo irá depender da quantidade de núcleos (*cores*) que o processador possui. Consequentemente, quanto mais núcleos um processador tem, mais tarefas podem ser executadas paralelamente (TEDESCO, 2020).

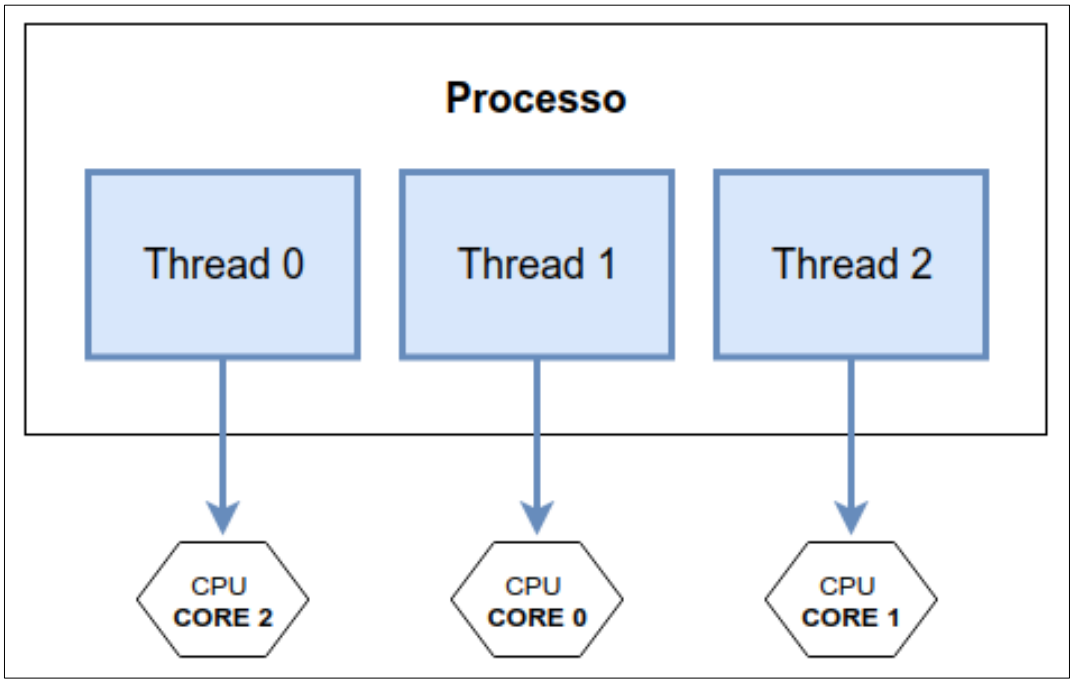


Figura 3: Exemplo de processo com três *threads* (In: TEDESCO, 2020)

Conforme Machado e Maia (p 84, 2017), o uso *threads* oferece a vantagem de poder “minimizar a alocação de recursos do sistema, além de diminuir o *overhead* na criação, troca e eliminação de processos”. Ademais, ao se tratar de desempenho, *threads* também contribuem para a diminuição do tempo de ociosidade da CPU (MEDEIROS, 2007).

Segundo Hennessy e Patterson (p C-2, 2019), o *pipelining* é uma técnica de implementação que torna os processadores rápidos. Com isso, as instruções recebidas no processador passam a ser sobrepostas, além de haver aproveitamento do paralelismo como um meio de realizar as operações necessárias que uma instrução depende para a sua execução. Portanto, o *pipelining* “é uma técnica que permite o processador executar múltiplas instruções paralelamente em estágios diferentes” (MACHADO; MAIA, 2017, p 29).

De acordo com Delgado e Ribeiro (p 427, 2017), a estrutura que compõe o *pipelining* é formada pela divisão de uma instrução em pequenas tarefas (também chamadas de “estágios”), onde cada tarefa (operação) será executada por uma unidade especializada por aquela operação referente a implementação do seu respectivo estágio. Este conjunto de unidades em linha é chamado de *pipeline* ou “cadeia de estágios”.

Conforme Machado e Maia (p 29, 2017), de modo semelhante ao que ocorre em uma linha de montagem, estas subtarefas, oriundas da instrução original, são compostas por “fases de busca da instrução e dos operandos, execução e armazenamento dos resultados”. Graças a esta técnica é possível que diferentes instruções estejam em fases distintas durante o processamento.

Na figura abaixo estão representados, o funcionamento sequencial (a) e o *pipeline* (c). Ambas apresentam quatro instruções que são subdivididas em três ciclos (B – Busca de instrução, D – Decodificação e E – Execução da instrução) em relação ao tempo. Cada ciclo, neste caso, possui uma operação que depende do resultado de seu antecessor para ser executado. Este resultado será temporariamente armazenado através do “registrador interestágio” (b) (DELGADO; RIBEIRO, 2017, p 427).

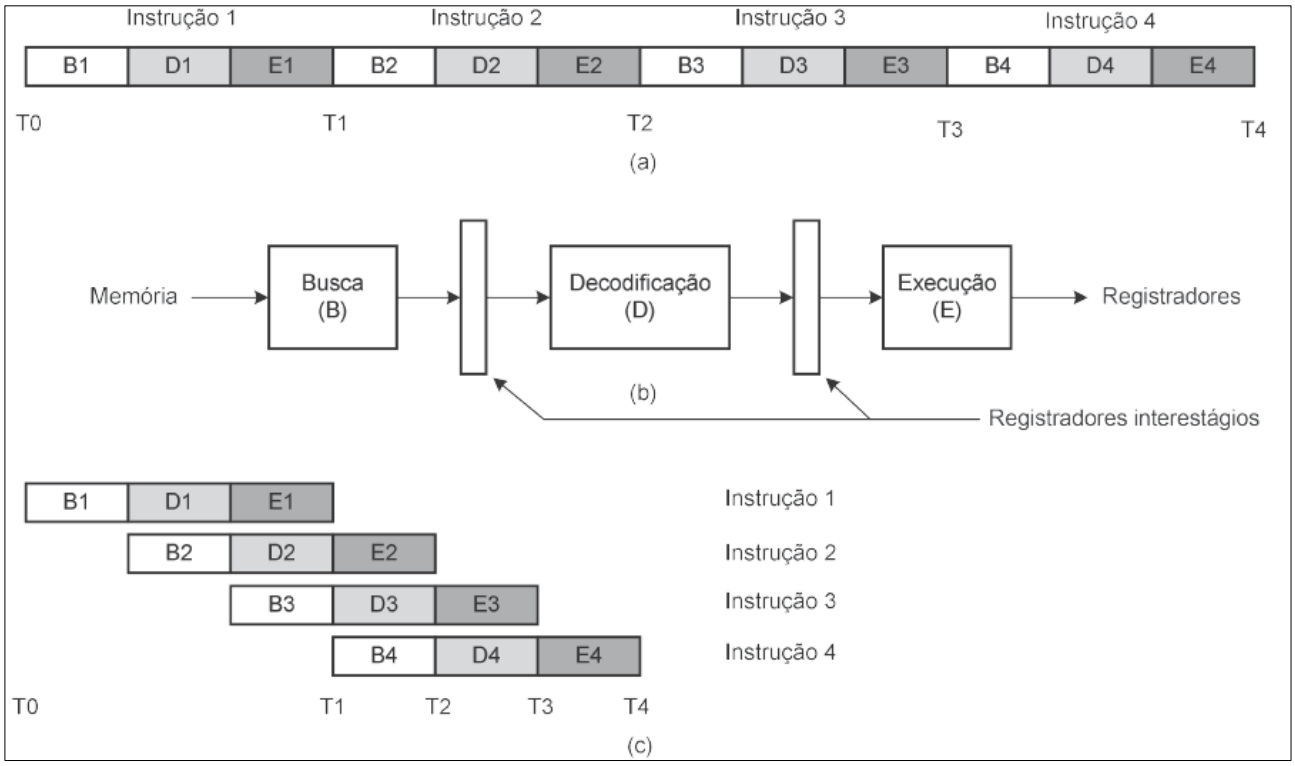


Figura 4: Diferença entre o funcionamento sequencial e o *pipeline* (In: DELGADO; RIBEIRO, p 427, 2017)

Diferente do que ocorre no funcionamento sequencial, o *pipeline* possibilita o processamento simultâneo de instruções e, devido a isso, a finalização de instruções passa a acontecer a cada ciclo ao invés de ser a cada três ciclos. Contudo, no *pipeline* as operações somente começam a ser executadas após a fase de “enchimento”, ou seja, apenas a partir do T1 será gerado o primeiro resultado de uma instrução. Em contrapartida, ao fim do *pipeline* inicia-se a fase de “esvaziamento”, isto é, deixam de entrar novas instruções para processamento, ocorrendo, portanto, a inatividade do *pipeline* após a finalização dos ciclos restantes (DELGADO; RIBEIRO, 2017, p 428).

4. PLATAFORMA JAVA

Segundo Schildt (p 3, 2015), a linguagem Java, criada em 1991 por James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan pela *Sun Microsystems*, a linguagem inicialmente se chamava “Oak”, porém, em 1995 a renomearam para “Java”. Para mais, o advento da linguagem se deve a uma necessidade da época em construir *softwares* embutidos para dispositivos eletrônicos domésticos sem que houvesse a necessidade de compilar o programa para cada especificação de CPU, como era o caso da linguagem C++, que, consequentemente, embora houvessem vários compiladores para esta linguagem, acabava por ser desvantajoso a construção de novos compiladores por causa do custo de tempo, dinheiro e por cada compilador ser destinado a um tipo de CPU específico.

Conforme Furgeri (p 13, 2015), outro fator que contribuiu para o seu desenvolvimento foi o surgimento da internet em 1993, onde a linguagem passaria a ser considerada também como uma plataforma de desenvolvimento, possibilitando assim, a sua utilização na construção de páginas para a *World Wide Web*.

Ademais, esta linguagem é multiplataforma, ou seja, um mesmo programa pode ser executado virtualmente em qualquer lugar sem que haja a preocupação de qual tipo de máquina o cliente possui (FURGERI, 2015, p 14).

Os programas codificados em Java geram uma linguagem intermediária denominada *bytecode* após a compilação e interpretação destes programas. Além de ser multiplataforma, o *bytecode* é interpretado a cada vez que o programa é executado por meio da JVM (*Java Virtual Machine*), máquina virtual necessária para a execução dos programas Java (SILVA; LEITE; OLIVEIRA, 2019, p 112).

O Java *Development Kit*, comumente conhecido como JDK, é uma ferramenta que agrega as bibliotecas, utilitários e o compilador da linguagem necessários para o desenvolvimento de programas (DIONISIO, 2013).

4.1. JAVA 8

Lançado pela Oracle em 2014, o Java 8 trouxe mais de 80 funcionalidades novas à linguagem e mudanças significativas na sua máquina virtual (JVM). Estas mudanças também permitiram uma nova forma de escrever códigos baseando-se conceitualmente em linguagens que fazem uso do paradigma funcional, oferecendo assim uma maior versatilidade e facilidade na programação de tarefas complexas e que antes demandavam muitas linhas de código (SILVA, 2016).

Uma das *features* introduzidas nesta versão da linguagem tem o nome de Expressões Lambda. A incorporação de tal recurso em Java possibilitou que métodos agora também possam receber comportamentos, ou seja, uma função como argumento de outra função (SILVA, 2016).

4.1.1. API DE *STREAM*

Como uma das funcionalidades incorporadas ao Java 8, a API de *Stream*, desenvolvida sob o pacote *java.util.stream*, disponibiliza recursos para manipular coleções de elementos de forma simplificada. Com isso tornou-se possível graças a adição do paradigma funcional em conjunto com as expressões lambda (SILVA, 2017).

De acordo com Silva (2017), para ser possível trabalhar com coleções com esta API, foi adicionado à API de *Collections* o método *stream* a fim de tornar estas coleções fontes de dados para *streams*.

Segundo Amorim (2015), uma *Stream* pode ser definida “como uma sequência de elementos de uma fonte de dados que suporta operações de agregação”, onde esta fonte de dados será processada e devolvida como uma nova coleção ou valor reduzido (*map-reduce*). Além disso, a API também fornece suporte para processamento de dados utilizando o paralelismo.

Para Silva (2016), uma *Stream* possui as seguintes definições:

- **Sequência de elementos:** Os elementos de uma *stream* não são armazenados, mas sim processados sob demanda.

- **Fonte de dados:** Além de permitir manipular coleções, a API também trabalha com dados oriundos de *arrays* e recursos de E/S (entrada e saída).
- **Operações de agregação:** Assim como em outras linguagens que suportam o paradigma funcional, algumas das operações disponibilizadas pela API de *Stream* envolvem a filtragem, modificação e transformação de um elemento para outro, tanto de forma serial quanto paralela.
- ***Pipeline*:** Este conceito torna mais flexível a manipulação de dados por intermédio da API, visto que ela proporciona criar uma cadeia de operações que irão formar um fluxo de processamento. Isto é possível pois parte das operações implementadas retornam novas *streams*.

  As operações dentro de um *pipeline* podem ser classificadas em intermediárias e terminais. A primeira diz respeito àquelas operações que "servem como entrada para outras operações intermediárias". A segunda se refere as operações que tem por finalidade fechar a cadeia de operações, encerrando o processo.
- **Iteração interna:** Diferentemente do que ocorre na iteração externa onde a implementação para percorrer os elementos de uma coleção é feita de forma explícita no código, na iteração interna a responsabilidade de percorrer os elementos e como manipulá-los fica a cargo da linguagem, incumbindo ao desenvolvedor somente o que ele deve fazer com estes elementos.

  Alguns exemplos de métodos que são aplicados este conceito são: *map()*, *forEach()* e *filter()*.

Dentro do escopo das operações presentes na API de *Stream* há particularidades importantes a serem destacadas, tais como:

- ***Lazy Evaluation*:** Métodos que são executados de forma *lazy*, ou seja, preguiçosa não realizam nenhum processamento até que uma operação terminal seja invocada (URMA; FUSCO; MYCROFT, 2019, p 94).

  Por meio desta característica é possível otimizar as operações de tal forma que não haja processamento desnecessário, resultando, por fim, na obtenção unicamente daquilo foi solicitado pela operação terminal, como é o caso das operações que retornam um *Optional*, presente em métodos como o *findAny* (SILVEIRA; TURINI, p 56-57).

Operações terminais como o *findAny* são implementados de forma inteligente, pois ao analisar as operações invocadas anteriormente, se dispensa a necessidade de filtrar todos os elementos da lista, visto que só é necessário que apenas um destes elementos cumpra o predicado. Contudo, esta técnica não garante a sua aplicabilidade em qualquer contexto, pois depende das operações contidas no *pipeline* (SILVEIRA; TURINI, p 58).

- ***Stateless*:** De acordo com Silva (2016), operações *stateless* são aquelas que não armazenam o estado dos elementos anteriores para manipular os elementos subsequentes que serão processados, ou seja, cada elemento independe do que acontece com os demais.

  Portanto, operações como *map* e *filter* são exemplos de *stateless*, no entanto a inexistência de um estado pressupõe que, ou não há tal presença na expressão atribuída para o método ou o método não possui um estado mutável interno (URMA; FUSCO; MYCROFT, 2019, p 115).
- ***Stateful*:** Segundo Silva (2016), operações que trabalham como *stateful* “podem incorporar o estado do elemento processado anteriormente no processamento de novos elementos”. Isto acontece em métodos como o *reduce*, *sum* e *max*, pois estes possuem um estado que vai acumulando o resultado obtido por cada elemento a fim de gerar o resultado final (URMA; FUSCO; MYCROFT, 2019, p 116).

  Para Silveira e Turini (p 59), no entanto, operações *stateful* “podem precisar processar todo o stream mesmo que sua operação terminal não demande isso”. Consequentemente isso resultaria em perda de performance, pois haveria processamento desnecessário.
- ***Short-circuiting*:** Esta otimização fez com que a API, encurtasse o processamento de uma *stream*, de tal modo que ela tenha somente com a parte necessária para a obtenção do resultado. Graças a isso, métodos como *allMatch*, *findFirst* e *limit* interrompem o fluxo de processamento assim que a sua condição é ou não atendida (URMA; FUSCO; MYCROFT, 2019, p 109).
- ***Reduction*:** Estas operações têm a finalidade de reduzir um conjunto de valores advindos de uma *stream* para um único valor e, portanto, são classificadas como operações terminais (URMA; FUSCO; MYCROFT, 2019, p 111).

Métodos como *average*, *count*, *min*, *max* e *sum* executam operações de redução. Algumas destas operações, no entanto, retornam um *Optional*, para casos em que não há um valor a ser retornado (SILVEIRA; TURINI, p 59).

- **Método *collect*:** De acordo com Silva (2016), o método *collect*, classificado como uma operação terminal, "possibilita coletar os elementos de uma stream na forma de coleções, convertendo uma stream para os tipos *List*, *Set* ou *Map*". Além disso, também é possível realizar outras operações, tais como o *mapping*, *collectingAndThen* e *grouppingBy* que concedem a combinação de elementos e o *joining* que tem a função de concatenar elementos para uma String. (SUBRAMANIAM, 2014, p 54).
- **Paralelismo:** A API também fornece suporte à paralelização por intermédio do método *parallelStream* que já contém toda a lógica de baixo nível necessária para a sua execução; deixando a cargo do desenvolvedor somente a construção do fluxo do *pipeline* (SILVA, 2016).

  Para Silveira e Turini (p 78), fica sob responsabilidade da API "decidir quantas threads deve utilizar, como deve quebrar o processamento dos dados e qual será a forma de unir o resultado final em um só."

  Contudo é preciso ter cautela, pois uma quantidade pequena de elementos pode ocasionar um *overhead*, que é "decorrente do processamento de tarefas adicionais geradas pela paralelização" (SILVA, 2016).

4.1.2. INTERFACES FUNCIONAIS

Em Java, Interfaces Funcionais possibilitam a escrita de códigos através do uso da sintaxe de expressões lambda. Esta abordagem garantiu que fosse mantida a compatibilidade com as versões anteriores da linguagem (SUBRAMANIAM, 2017).

De acordo com Subramaniam (2017), as características de uma interface funcional são:

- Possuir somente um método abstrato, porém, há uma ressalva para métodos abstratos que forem públicos e que estejam na classe *Object*. Para estes casos, os métodos não serão contabilizados na interface.
- Há interface pode conter métodos *default* e estáticos.

Ademais, para que o compilador possa reconhecer que uma interface é realmente funcional, a linguagem disponibilizou a anotação *@FunctionalInterface*¸ para que o compilador gere algum erro de compilação caso os requisitos para a criação desta interface não sejam respeitados (CARVALHO, 2018).

4.2. JAVA 18 e 19

Atualmente, a versão mais recente disponibilizada é o Java 18, que “oferece milhares de melhorias de desempenho, estabilidade e segurança, incluindo nove melhorias na plataforma que impulsionarão ainda mais a produtividade do desenvolvedor”. Algumas das atualizações e melhorias presentes nesta versão envolvem a definição por padrão do UTF-8 que visa garantir o comportamento esperado pelas implementações, localidades e configurações. Além disso, foi adicionada uma ferramenta para trabalhar com um servidor web simples através de linha comando, reimplementação de algumas classes que trabalham com a API de *reflection* (INFORCHANNEL, 2022).

A próxima versão da linguagem (JDK 19), prevista para chegar em setembro de 2022, pretende avançar com a capacidade de interoperar código fora do Java *runtime* por meio de uma API de função e memória externa, além de outras propostas que englobam *generics* até uma porta RISC-V. Outra possibilidade de inclusão seria a uma API de vetor para cálculos vetoriais que compilariam em tempo de execução (KRILL, 2022).

5. PROPOSTA DO TRABALHO

Este trabalho tem como proposta analisar o comportamento e o desempenho da API de *Stream* do Java tanto em seu modo serial quanto paralelo, a partir de implementações que serão processadas nestes dois modos. Os resultados que serão obtidos por meio destes testes têm como critérios de análise o tempo de processamento (milissegundos), o pico de uso da CPU (%), o consumo de memória (*megabytes*) e a quantidade de *threads* utilizada durante a execução dos métodos. Por fim, com base nos dados coletados, pode-se concluir em qual modo de processamento a API se mostra mais performática em comparação ao outro modo.

Para proceder com a execução dos testes, a estrutura para o desenvolvimento deste trabalho consiste na criação de uma base de dados que é a fonte para o processamento dos métodos que estão implementados em Java, com a finalidade de garantir a consistência e fidedignidade nos resultados gerados.

Os métodos, por sua vez, são implementados com menor e maior grau de complexidade e exigência do processador com o propósito de averiguar qual o comportamento durante a execução e como esta complexidade pode afetar o desempenho do processamento, bem como os possíveis diferentes resultados oriundos destas variações.

Para cada método a ser processado, há uma quantidade exata de dados a serem utilizados durante os testes, sendo eles: 100 (cem), 1.000 (mil), 10.000 (dez mil) e 100.000 (cem mil) e, em cada uma destas ocasiões, o método é executado 3 (três) vezes com o objetivo de obter resultados mais precisos. Ao final de cada conjunto de 3 (três) testes há o cálculo da média dos resultados de tempo de processamento e o pico de uso da CPU.

Além disso, no processamento de cada variação da quantidade de dados é aplicado tanto o modo serial quanto o paralelo da API de *Stream*. Portanto, a totalização de execuções por método e de acordo com o modo escolhido para processar estes dados é de 12 testes.

Os testes são monitorados por meio da ferramenta VisualVM e os dados coletados estão salvos em um arquivo Excel e também em um arquivo gerado pela própria ferramenta, caso seja necessária uma consulta mais detalhada no futuro.

5.1. TECNOLOGIAS E RECURSOS UTILIZADOS

5.1.1. MAVEN

De acordo com Ottero (2012), Maven é uma ferramenta utilizada para gerenciamento e automação de construção de projetos. Além disso, ele também oferece recursos adicionais por meio de *plugins*. Desta maneira torna-se mais prático a organização, desenvolvimento e manutenção de projetos.

5.1.2. LOMBOK

Lombok é um *framework* que, por meio do uso de anotações, abstém o programador de escrever certos códigos ou funcionalidades que comumente se repetem em um projeto, tais como os *Getters* e *Setters*, *HashCode* e *Equals*, entre outros. Dito isso, estes códigos que ficarão omitidos, serão gerados automaticamente de acordo com a anotação invocada (SOUZA, 2013).

5.1.3. VISUALVM

Distribuída junto a JDK a partir da versão 6, a ferramenta VisualVM possibilita monitorar em tempo real aplicações desenvolvidas em Java por meio de gráficos e relatórios. Neste monitoramento é possível consultar informações referentes ao tempo de processamento de métodos, ao uso da CPU e memória (FACHOLI, 2015).

5.1.4. PYTHON

Segundo Divino (2021), “Python é uma linguagem de programação interpretada de alto nível e que suporta múltiplos paradigmas de programação: imperativo, orientado a objetos e funcional. É uma linguagem com tipagem dinâmica e gerenciamento automático de memória”.

5.1.5. FAKER

Para Sahu (2021), Faker é uma biblioteca em Python utilizada para gerar dados falsos. Esta biblioteca já possui uma gama de funções que retornam diversos tipos de dados, como nomes, endereços, locais e data de nascimento falsos.

6. DESENVOLVIMENTO DO TRABALHO

A implementação deste projeto está dividida em duas etapas, sendo a primeira responsável por gerar a base de dados e a segunda por fazer uso dos dados gerados para analisar o comportamento e desempenho por intermédio dos métodos implementados.

O computador onde os testes foram processados possui a seguinte configuração: sistema operacional Windows 10 *Home Single Language*, processador Intel *Core* i5-7200U 2,70GHz, memória RAM de 8Gb e SSD de 240Gb SATA III.

6.1. GERAÇÃO DA BASE DE DADOS

A base de dados usada durante os testes tem a sua implementação desenvolvida por meio da linguagem Python em conjunto com a biblioteca Faker para fornecer, de forma mais simplificada e ágil, os dados fictícios.

O contexto definido para a geração de dados é o de Pessoa, que possui as seguintes informações: nome completo, data de nascimento, etnia, estado civil, profissão, tipo de pessoa (CPF ou CNPJ), salário e telefone.

Para que a aplicação Java possa fazer uso destes dados, as informações das pessoas foram adicionadas em um arquivo criado no formato *.txt*, onde cada linha deste arquivo consiste nos dados de uma pessoa e cada informação desta pessoa tem o separador (;) com a finalidade de auxiliar a programação desenvolvida em Java na atribuição de cada informação no respectivo atributo da classe Pessoa.

No código da figura 5 é definido o arquivo e o tipo de operação realizada sobre ele, que no caso é de escrita de dados, e logo após a criação da lista que armazena as informações de pessoas. Posteriormente é executado um *loop* onde cada iteração representa as informações de uma pessoa fictícia e cada pessoa criada será adicionada à lista de pessoas. Por último, o conteúdo desta lista é escrito no arquivo e a operação é encerrada.

```python
arquivo = open('base-dados.txt', 'w', encoding = "utf-8")

pessoas = list()

for randomicNumber in range(1_000_000):
  pessoa = fake.first_name() + ' ' + fake.last_name() \
    + SEPARADOR + str(fake.date_of_birth()) + SEPARADOR
  pessoa += fake.etnia().value + SEPARADOR
  pessoa += fake.estadoCivil().value + SEPARADOR
  pessoa += gerarStringDeProfissaoComCNPJOuCPF(randomicNumber)
  pessoa += gerarStringComSalarioDaPessoa()
  pessoa += gerarStringComTelefoneDaPessoa(randomicNumber)
  pessoa += fake.country() + SEPARADOR
  pessoa += '\n'

  pessoas.append(pessoa)

arquivo.writelines(pessoas)
arquivo.close()
```

Figura 5: Código em Python para geração dos dados

O link para consulta do código fonte da geração da base de dados está disponível em: <https://colab.research.google.com/drive/1dT3Bc3WiJ899aP5Cio7JOeQzVmOa8UcK?usp=sharing>.

6.2. IMPLEMENTAÇÃO DO PROJETO EM JAVA

Para desenvolvimento da aplicação em Java, foi utilizado o Maven como gerenciador de dependências com o intuito de facilitar a adição de outras tecnologias no projeto, como é o caso do Lombok, tornando-o assim mais flexível e simples a sua execução em outros computadores.

A implementação deste projeto consome os dados oriundos da aplicação desenvolvida em Python – responsável por gerar os dados de pessoas – e estas informações passam a ser mantidas em uma lista do tipo Pessoa na memória ao executar a aplicação.

```java
@Getter
@Setter
@Builder
@AllArgsConstructor(access = AccessLevel.PRIVATE)
@ToString
public class Pessoa {

    private String nome;
    private LocalDate dataNascimento;
    private Etnia etnia;
    private EstadoCivil estadoCivil;
    private String profissao;
    private String cpf;
    private String cnpj;
    private Integer salario;
    private String telefone;
    private String paisDeOrigem;
}
```

Figura 6: Classe Pessoa

A figura acima (6) representa a classe que possui os atributos referentes a Pessoa para manipulação durante os testes. Nela também há as anotações do Lombok, para facilitar e agilizar o desenvolvimento da aplicação, e com isso reduzir a estrutura da classe.

Na figura 7, é apresentado o método responsável por extrair os dados de uma pessoa durante a leitura do arquivo, onde cada linha dele corresponde as informações de uma pessoa.

O separador utilizado durante a geração dos dados passa a ser usado neste código para separar os respectivos dados ao atributo correspondente da classe Pessoa. Ao final da execução do método é retornado uma Pessoa para que seja, posteriormente, adicionada a lista que contém os dados para serem processados pelos métodos implementados para testes.

```java
private static Pessoa extrairDadosDaPessoaDoArquivo(String dadosPessoa) {
    final String SEPARADOR_DADOS = ";";
    String[] dadosPessoais = dadosPessoa.split(SEPARADOR_DADOS);
    int indiceDadosPessoais = 0;

    PessoaBuilder pessoaBuilder = Pessoa.builder()
            .nome(dadosPessoais[indiceDadosPessoais])
            .dataNascimento(converterDataNascimentoParaLocalDate(dadosPessoais[++indiceDadosPessoais]))
            .etnia(ConvertToEnum.converter(Etnia.class, dadosPessoais[++indiceDadosPessoais]))
            .estadoCivil(ConvertToEnum.converter(EstadoCivil.class, dadosPessoais[++indiceDadosPessoais]));

    boolean pessoaJuridica = Boolean.valueOf(dadosPessoais[++indiceDadosPessoais].toLowerCase());

    pessoaBuilder.profissao(dadosPessoais[++indiceDadosPessoais])
            .cnpj(pessoaJuridica ? dadosPessoais[++indiceDadosPessoais] : null)
            .cpf(!pessoaJuridica ? dadosPessoais[++indiceDadosPessoais] : null)
            .salario(Integer.parseInt(dadosPessoais[++indiceDadosPessoais]))
            .telefone(dadosPessoais[++indiceDadosPessoais])
            .paisDeOrigem(dadosPessoais[++indiceDadosPessoais]);

    return pessoaBuilder.build();
}
```

Figura 7: Método para gerar uma Pessoa

Há também uma classe que é composta pelos métodos utilizados no decorrer dos testes, bem como a configuração definida previamente, constituída pelo tipo da Stream e o tamanho da base de dados processada, como consta no **APÊNDICE A –** Classe TesteStreamPessoa.

Ao todo, esta classe dispõe de 4 métodos implementados para a realização dos testes e análises. Ademais, cada método possui uma complexidade e finalidade de processamento distinta uma das outras, como é apresentado adiante.

```java
public void buscarMaiorSalarioDePessoasJuridicasCasadasPorPais() {
    Map<String, Optional<Pessoa>> collect = baseStreamPessoa
        .filter(pessoa -> pessoa.getEstadoCivil().equals(EstadoCivil.CASADO))
        .collect(Collectors.groupingBy(Pessoa::getPaisDeOrigem,
                Collectors.filtering(pessoa -> Objects.nonNull(pessoa.getCnpj()),
                        Collectors.maxBy(Comparator.comparing(Pessoa::getSalario)))));
}
```

Figura 8: Implementação do método buscarMaiorSalarioDePessoasJuridicasCasadasPorPais

A implementação do método acima processa a lista de pessoas com a finalidade de buscar as pessoas jurídicas que sejam casadas e que possuam o maior salário de seu país. Portanto, incialmente é feito uma filtragem para encontrar aquelas pessoas que atendam a condição de casadas. Logo após isso estas pessoas são agrupadas com base no país onde

moram, desde que elas possuam um CNPJ. Por último há uma comparação para encontrar o maior salário por país.

```java
public void filtrarPessoasPeloSalario() {
    List<Pessoa> pessoas = baseStreamPessoa
            .filter(pessoa -> pessoa.getSalario() > 2000)
            .collect(Collectors.toList());
}
```

Figura 9: Implementação do método filtrarPessoasPeloSalario

Esta segunda implementação tem a função de filtrar as pessoas que possuam um salário maior que 2.000 e retornar o resultado no formato de lista com as pessoas que atenderam a condição determinada.

```java
public void buscarPessoaQuePossuiMenorSalario() {
    Optional<Pessoa> pessoaComMenorSalario = baseStreamPessoa
            .min(Comparator.comparing(Pessoa::getSalario));
}
```

Figura 10: Implementação do método buscarPessoaQuePossuiMenorSalario

O método apresentado na figura 10 tem o propósito de encontrar na lista de pessoas aquela que possui o menor salário.

```java
public void agruparMediaSalarialDePessoasPorPais() {
    Map<String, Double> mediaSalarialDePessoasPorPais = baseStreamPessoa
        .collect(Collectors.groupingBy(Pessoa::getPaisDeOrigem,
                Collectors.averagingInt(Pessoa::getSalario)));
}
```

Figura 11: Implementação do método agruparMediaSalarialDePessoasPorPais

Nesta última implementação (figura 11), o objetivo é calcular a média salarial das pessoas de acordo com o seu país e guardar o resultado conforme a nação correspondente.

Link para consulta do código fonte dos métodos implementados: <https://github.com/paulocromano/comparacao-stream-serial-paralela-java>.

6.3. RESULTADOS

6.3.1. MÉTODO buscarMaiorSalarioDePessoasJuridicasCasadasPorPais

DADOS PROCESSADOS: 100

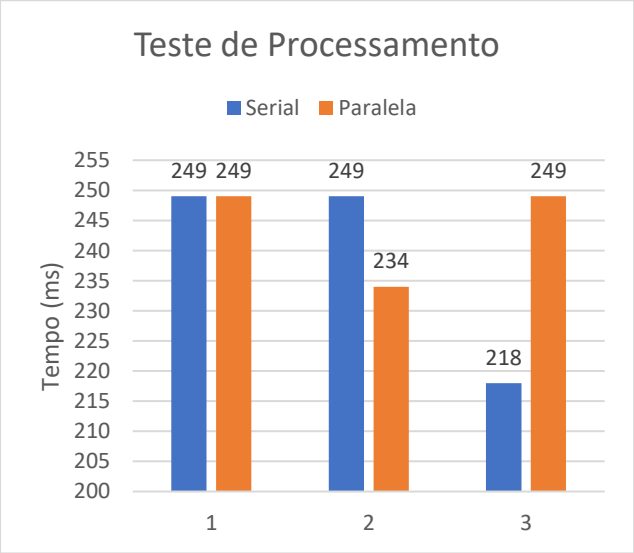


Figura 12: Método 1 – Teste de processamento com 100 dados

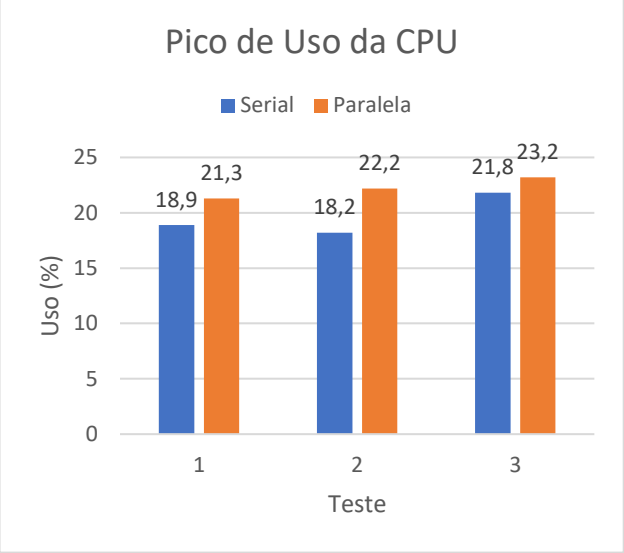


Figura 13: Método 1 – Pico de uso da CPU com 100 dados

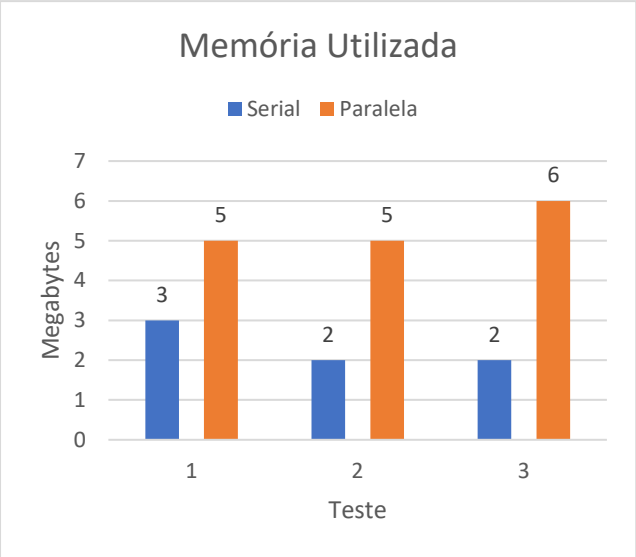


Figura 14: Método 1 – Memória utilizada com 100 dados

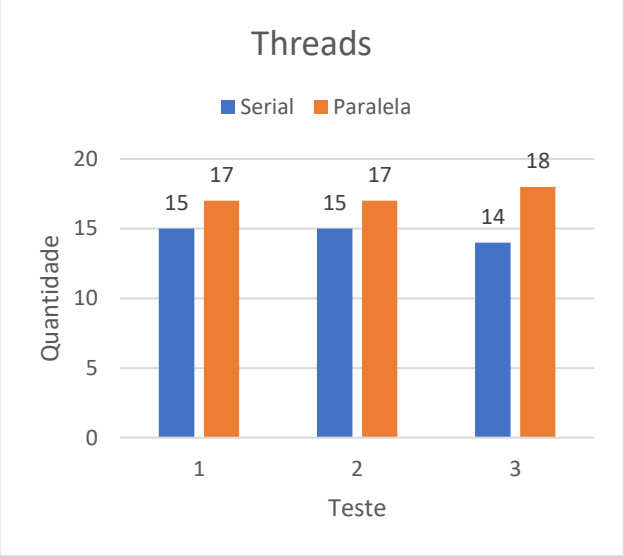


Figura 15: Método 1 – *Threads* com 100 dados

A partir dos resultados obtidos do tempo de processamento, nota-se que durante o primeiro teste houve um empate, enquanto no segundo a *stream* serial levou 15ms a mais de processamento que a paralela. Já o terceiro teste ocorreu a situação inversa, onde o modo paralelo teve um tempo superior de 31ms.

Quanto ao pico de uso da CPU, em todos os cenários a *stream* paralela teve um consumo maior que a serial, com variação entre 1,4% a 4%.

Na utilização de memória, a *stream* paralela esteve a frente em todos os testes com uma diferença mínima de 2Mb e máxima de 4Mb.

O cenário também se repetiu no uso de *threads*, onde a *stream* paralela fez uso de 2 a 4 *threads* a mais que o modo serial.

DADOS PROCESSADOS: 1.000

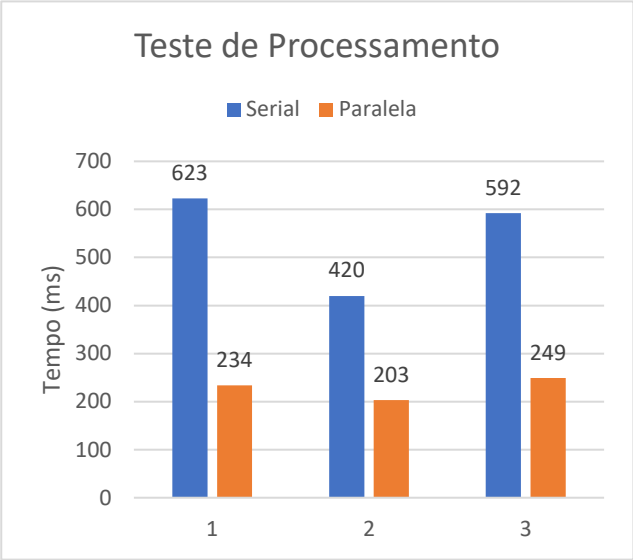


Figura 16: Método 1 – Teste de processamento com 1.000 dados

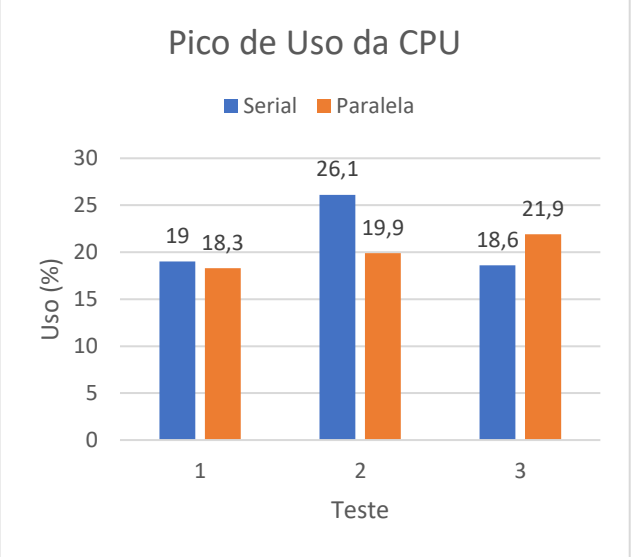


Figura 17: Método 1 – Pico de uso da CPU com 1.000 dados

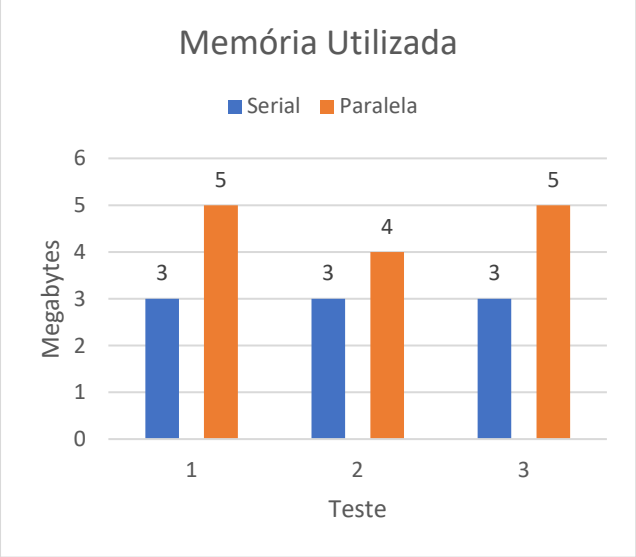


Figura 18: Método 1 – Memória utilizada com 1.000 dados

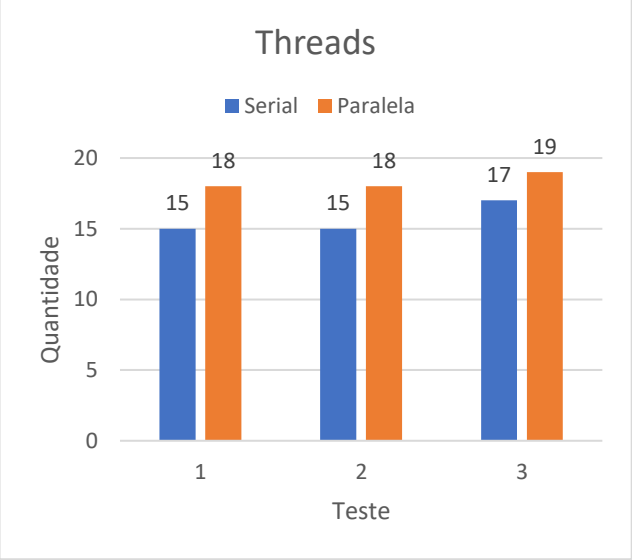


Figura 19: Método 1 – Threads com 1.000 dados

Com o aumento de dados a serem processados, ocorreu um aumento maior com relação ao tempo de processamento em comparação aos testes com 100 dados. Neste caso, a

stream paralela se sobressaiu em todos os cenários, com uma vantagem mínima de 217ms e com um pico de 343ms.

O pico de uso da CPU foi minimamente maior no primeiro teste, onde este custo ficou por conta da *stream* serial com 0,7% de diferença. Nos dois testes subsequentes o modo paralelo se destacou com divergência de 6,2% e 3,3%, respectivamente.

A maior utilização de memória também se deu por parte da forma de processamento paralela com dissemelhança entre 1Mb e 2Mb.

Por fim, a quantidade de *threads* usada para processar estes dados pela *stream* serial foi inferior nos três testes, com 2 a 3 *threads* a menos que a paralela.

QUANTIDADE DE DADOS: 10.000

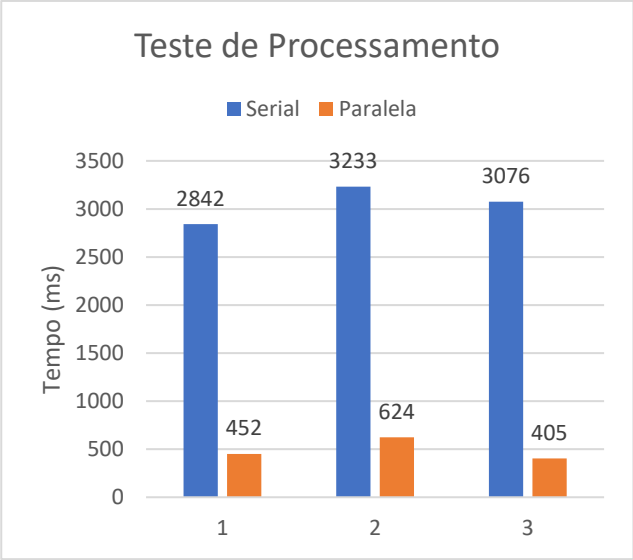


Figura 20: Método 1 – Teste de processamento com 10.000 dados

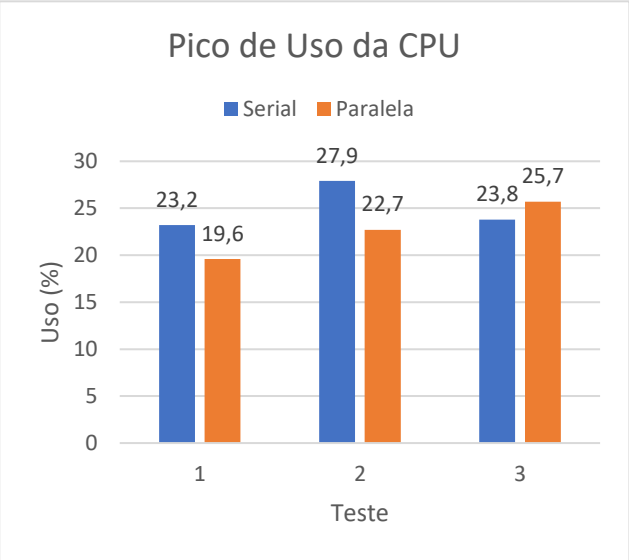


Figura 21: Método 1 – Pico de uso da CPU com 10.000 dados

Neste ambiente de testes, o tempo de processamento pela *stream* serial teve um aumento exponencial em comparação à paralela, com uma diferença de até 2.671 milissegundos.

O da CPU obteve maior pico nos dois primeiros testes pelo modo serial, com 3,6% e 5,2% respectivamente a mais que o paralelo. Já no terceiro teste a *stream* paralela alcançou um pico maior, sendo este de 1,9% em relação ao outro modo.

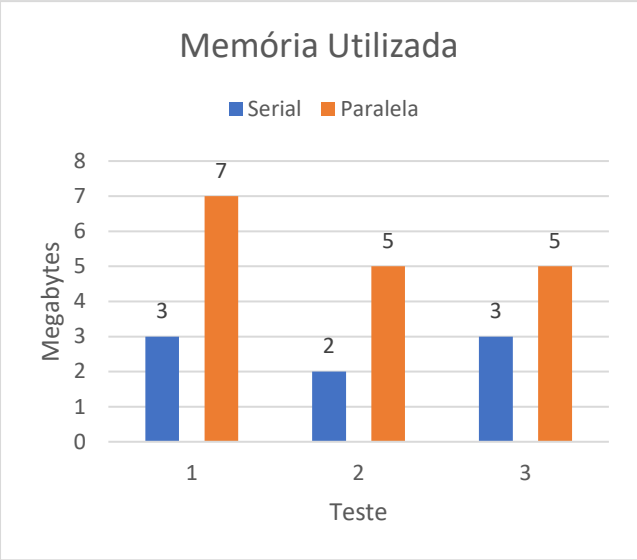


Figura 22: Método 1 – Memória utilizada com 10.000 dados

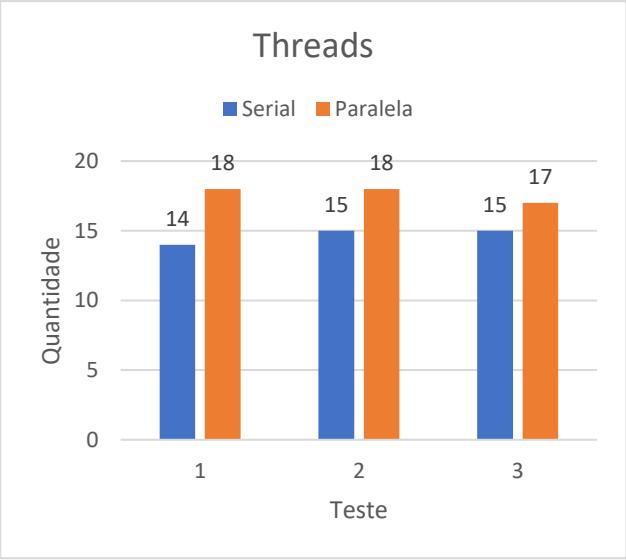


Figura 23: Método 1 – Threads com 10.000 dados

Nestas duas análises, o processamento com a *stream* paralela se sobressaiu em todos os aspectos. Na utilização da memória com valores maiores de 4Mb, 3Mb e 2Mb, respectivamente. Já no uso de *threads*, precisou de até 4 a mais em comparação a *stream* serial.

QUANTIDADE DE DADOS: 100.000

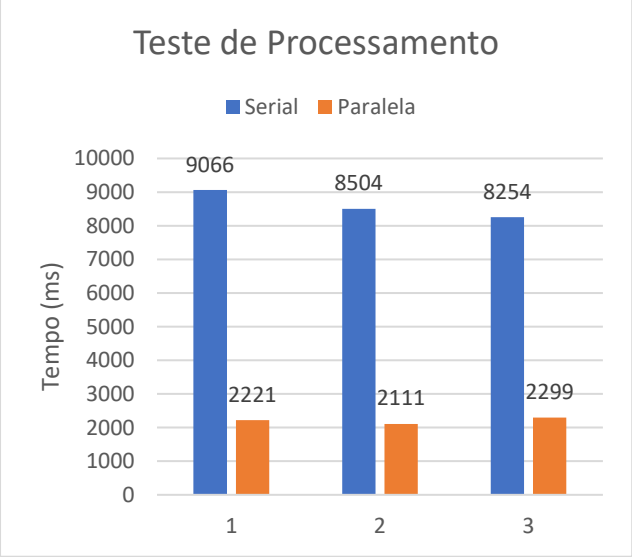


Figura 24: Método 1 – Teste de processamento com 100.000 dados

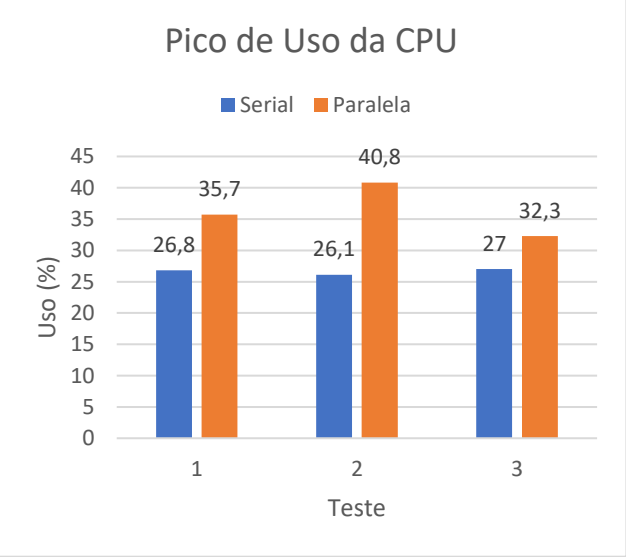


Figura 25: Método 1 – Pico de uso da CPU com 100.000 dados

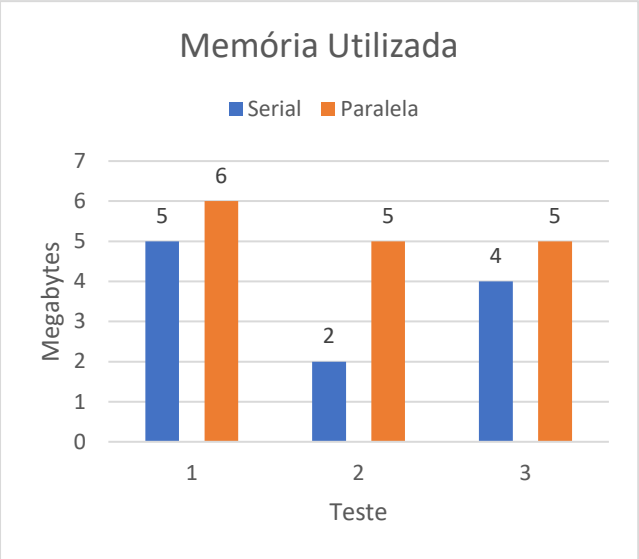


Figura 26: Método 1 – Memória utilizada com 100.000 dados

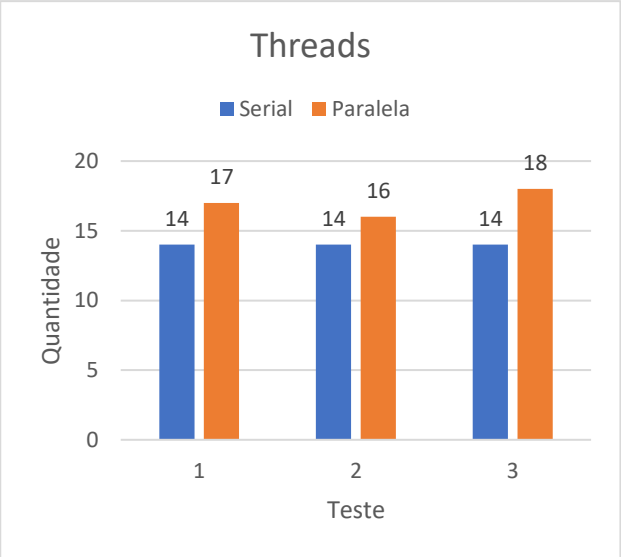


Figura 27: Método 1 – Threads com 100.000 dados

Assim como nos resultados apresentados no processamento com 10.000 dados, neste caso a *stream* serial também obteve um desempenho inferior à paralela, porém, com um tempo ainda pior de 5955ms e com a maior duração dentre todos os testes realizados neste método, alcançando, portanto, 6845ms.

Quanto ao pico da CPU, o modo paralelo atingiu o maior patamar dentre os demais testes: 40,8%. A diferença de uso entre cada teste alcançou os seguintes valores: 8,9%, 14,7% e 5,3%.

A utilização da memória durante os testes foi levemente superior por parte da *stream* paralela no primeiro e terceiro, com a diferença de apenas 1Mb e de 3Mb no segundo.

No resultado do uso de *threads* houve pouca dissemelhança, sendo elas: 3, 2 e 4, de acordo com a ordem dos testes, porém com o mesmo resultado dos testes realizados com 10.000 dados, ou seja, o processamento por meio da *stream* paralela fez maior uso de *threads*.

6.3.2. MÉTODO filtrarPessoasPeloSalario

QUANTIDADE DE DADOS: 100

A *stream* paralela durante os dois primeiros testes de tempo de processamento se mostrou menos eficiente do que a *stream* serial. O tempo a mais resultou em 15,7ms e 0,1ms de diferença. Já no último teste, o modo serial deu um grande salto em relação ao paralelo, com quase o dobro de discrepância: 31,1ms.

Por meio da análise dos resultados do pico de uso da CPU, nota-se que o modo serial fez uma maior utilização no primeiro e segundo teste, sendo a maior diferença no segundo teste com 6,7%. Em contrapartida, o último teste utilizou-se 1,9% mais a CPU com a *stream* paralela.

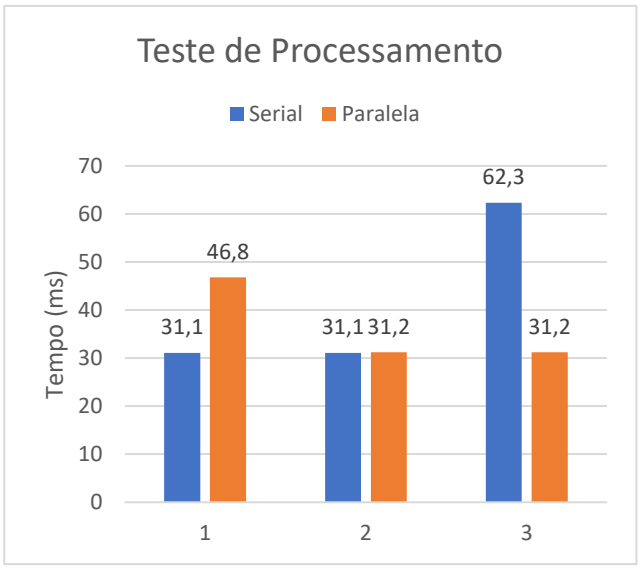


Figura 29: Método 2 – Teste de processamento com 100 dados

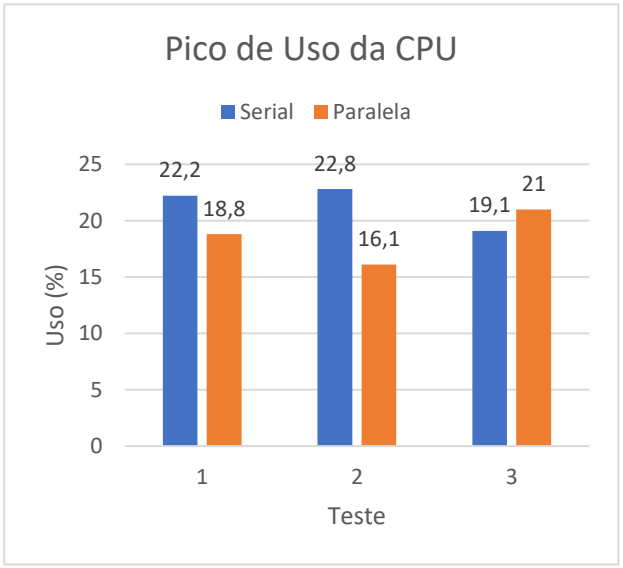


Figura 28: Método 2 – Pico de uso da CPU com 100 dados

Os resultados apresentados quanto ao uso de memória e *threads* levam, por unanimidade, uma requisição maior por parte da *stream* paralela. No primeiro caso ocorreu uma variação entre 2Mb e 3Mb. Já no segundo, a quantidade de *threads* usada a mais em comparação ao modo serial variou entre 3 e 4.

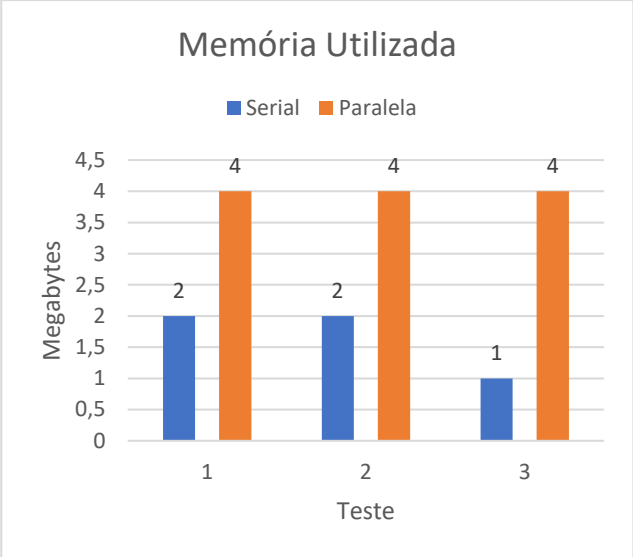


Figura 30: Método 2 – Memória utilizada com 100 dados

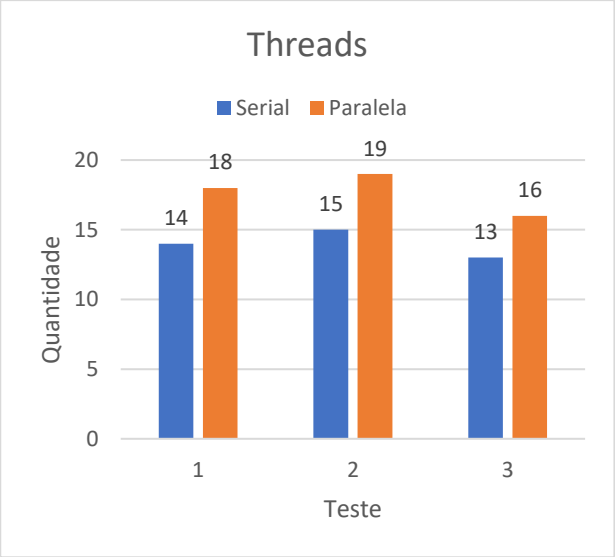


Figura 31: Método 2 – Threads com 100 dados

QUANTIDADE DE DADOS: 1.000

Constatou-se por meio dos resultados obtidos por esta quantidade de dados que o tempo de processamento foi maior por parte do modo serial em todos os testes. A menor diferença atingida transcorreu no segundo teste, onde a *stream* paralela conseguiu uma vantagem de 107,2ms, já a maior disparidade aconteceu no primeiro teste: 310,8ms.

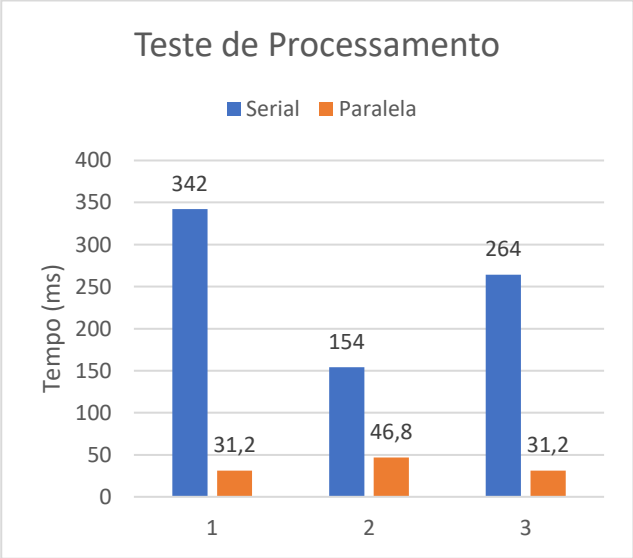


Figura 32: Método 2 – Teste de processamento com 1.000 dados

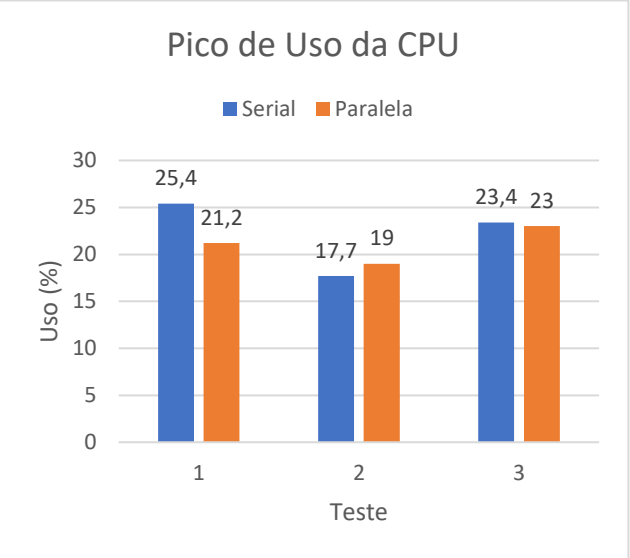


Figura 33: Método 2 – Pico de uso da CPU com 1.000 dados

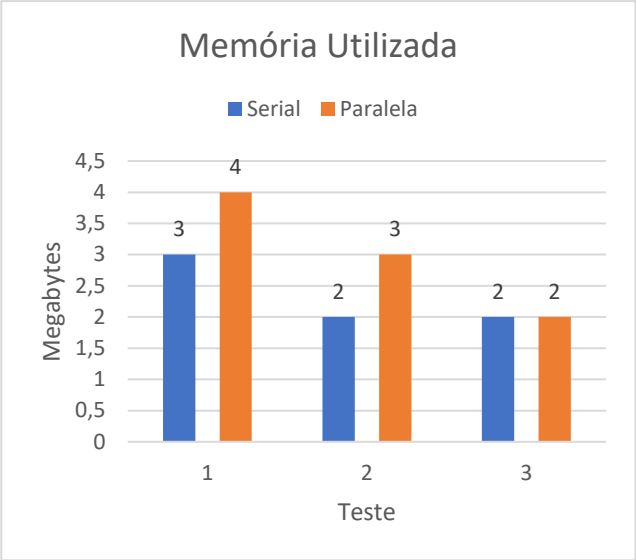


Figura 34: Método 2 – Memória utilizada com 1.000 dados

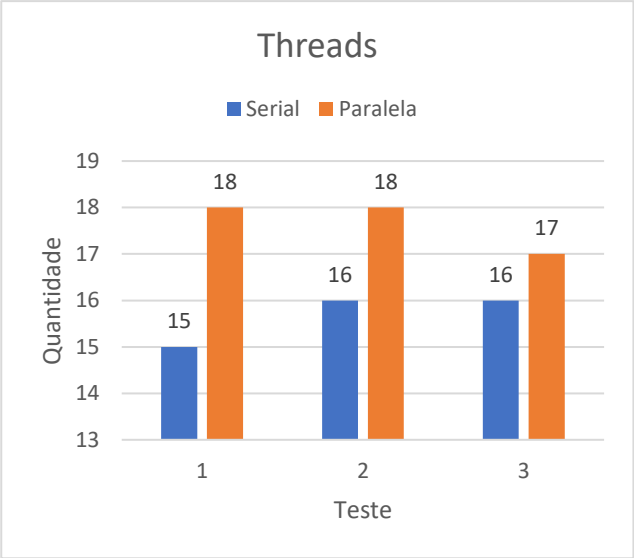


Figura 35: Método 2 – Threads com 1.000 dados

No uso de memória, o modo serial levou vantagem nos dois primeiros testes. Seu consumo foi de 1Mb a menos em ambos os testes. Já no último, houve um empate de 2Mb utilizados.

A quantidade de *threads* teve mais utilização com a *stream* paralela, com 3, 2 e 1 *threads* a mais, respectivamente, que a serial.

QUANTIDADE DE DADOS: 10.000

Na análise do tempo de processamento é nítida a vantagem que a *stream* paralela leva sobre a serial, sendo até 2378ms mais rápida. Por outro lado, o uso da CPU atingiu um pico maior através do modo serial, sendo utilizada até 4,2% a mais.

Quanto ao uso de memória, o maior consumo em todos os testes se deu pela *stream* paralela: 2Mb, 3Mb e 3Mb, respectivamente. Este cenário se assemelha quanto a utilização de *threads*, onde o modo serial teve um uso superior no primeiro e terceiro teste, enquanto no segundo houve um empate.

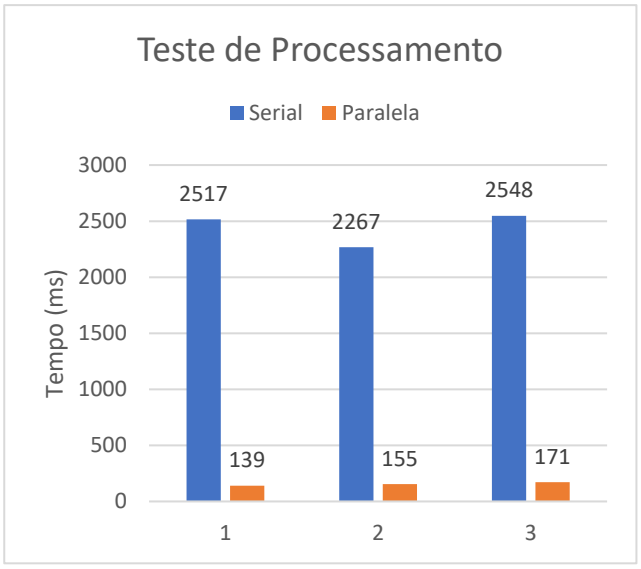


Figura 36: Método 2 – Teste de processamento com 10.000 dados

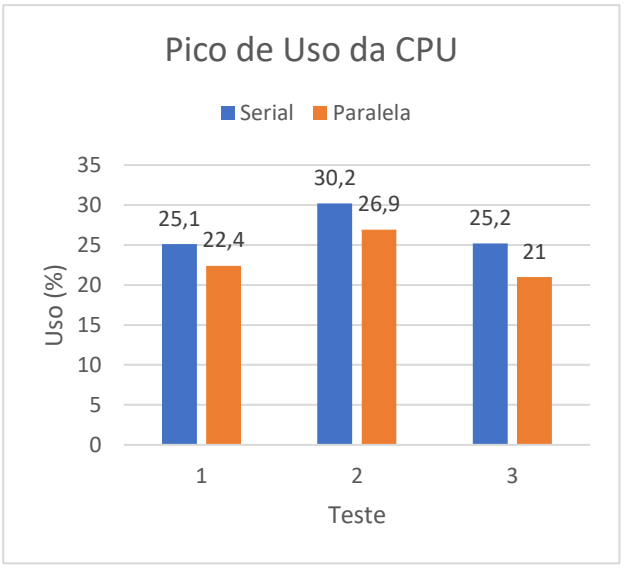


Figura 37: Método 2 – Pico de uso da CPU com 10.000 dados

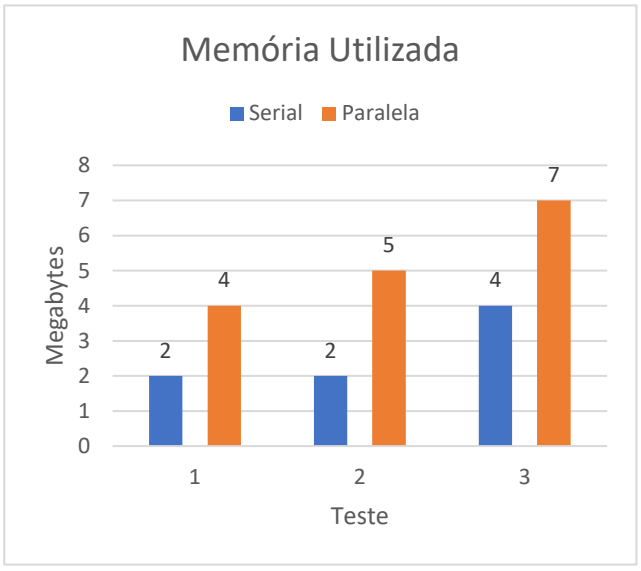


Figura 38: Método 2 – Memória utilizada com 10.000 dados

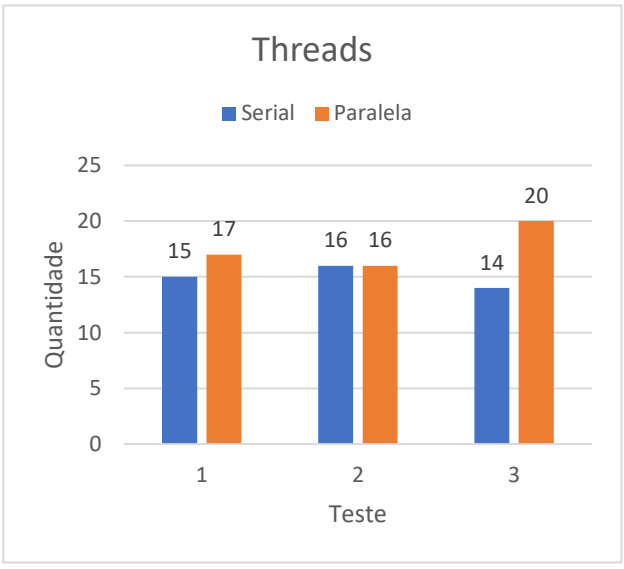


Figura 39: Método 2 – Threads com 10.000 dados

QUANTIDADE DE DADOS: 100.000

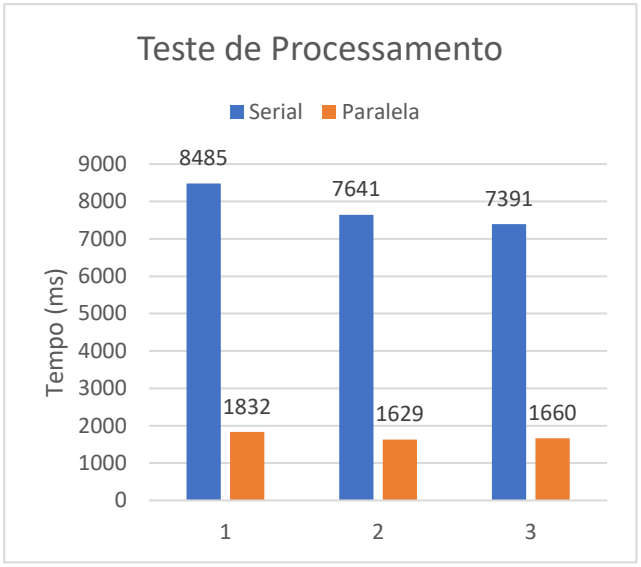


Figura 40: Método 2 – Teste de processamento com 100.000 dados

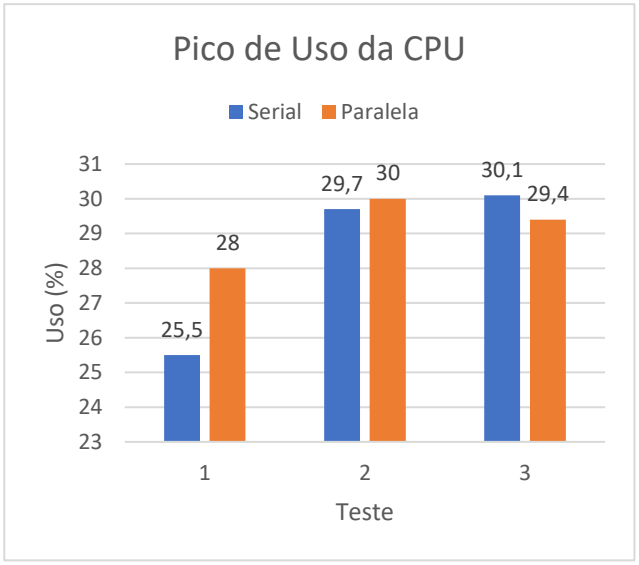


Figura 41: Método 2 – Pico de uso da CPU com 100.000 dados

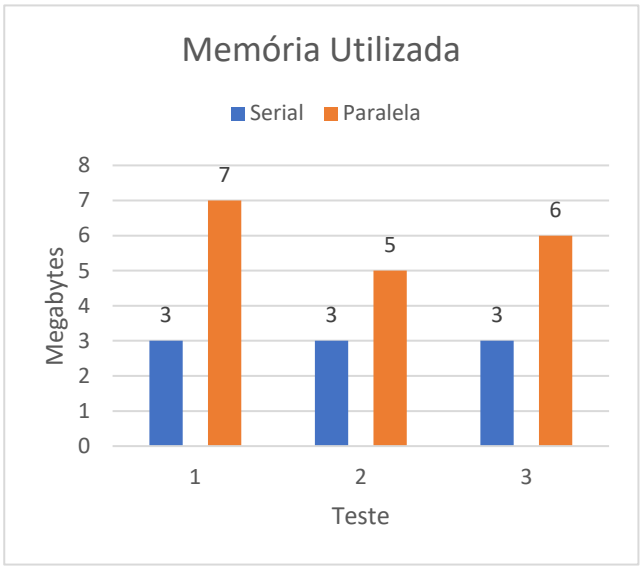


Figura 42: Método 2 – Memória utilizada com 100.000 dados

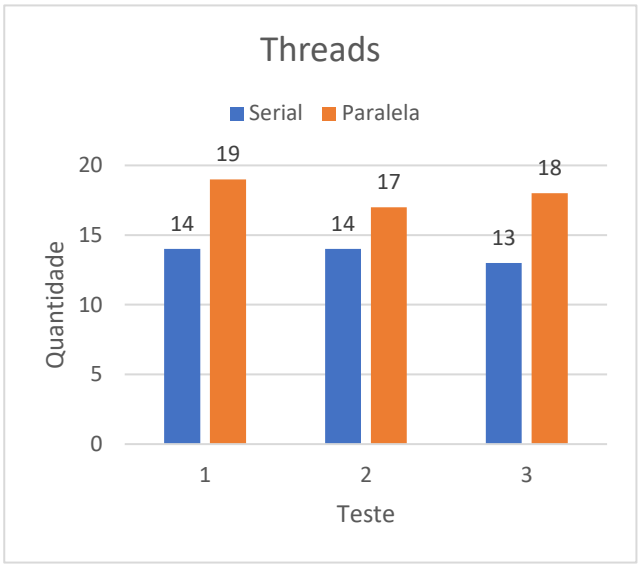


Figura 43: Método 2 – Threads com 100.000 dados

O tempo de processamento, assim como nos resultados obtidos com 10.000 dados, perdurou por mais tempo a *stream* serial com uma larga diferença que variou entre 5731ms a 6653ms.

O pico atingido pelo uso da CPU teve o seu ápice nos dois primeiros testes pelo modo paralelo, com a maior diferença de 2,5% entre estes dois testes. Não obstante, o terceiro teste apresentou a situação inversa, onde o modo serial teve um pico de 30,1% e, portanto, 0,7% a mais que o paralelo.

O destaque também vai para o modo paralelo em relação a quantidade de *threads*, visto que este modo ficou à frente em todos os testes com uma margem de 3 a 5 a mais.

6.3.3. MÉTODO buscarPessoaQuePossuiMenorSalario

QUANTIDADE DE DADOS: 100

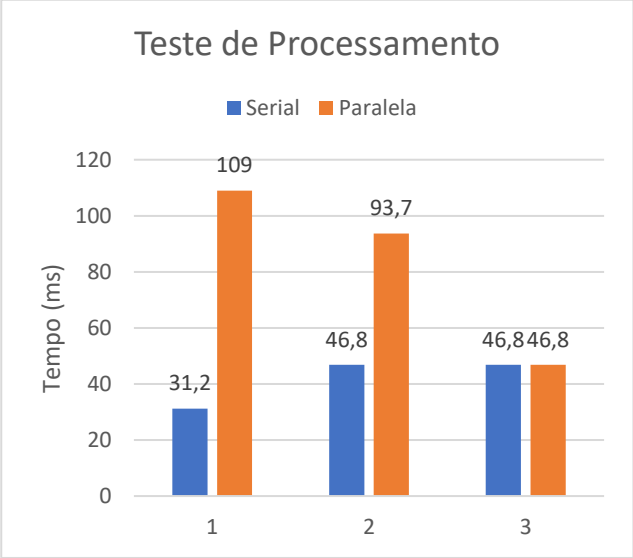


Figura 44: Método 3 – Teste de processamento com 100 dados

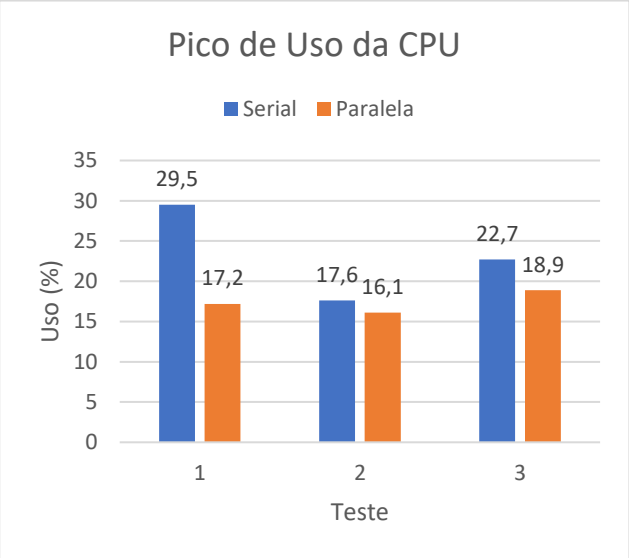


Figura 45: Método 3 – Pico de uso da CPU com 100 dados

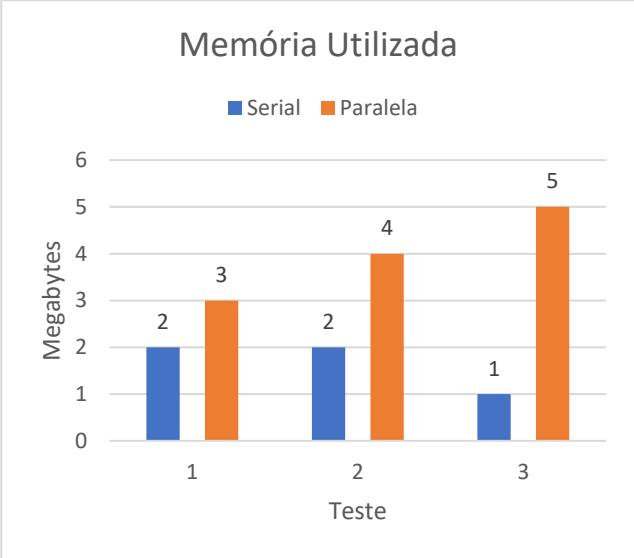


Figura 46: Método 3 – Memória utilizada com 100 dados

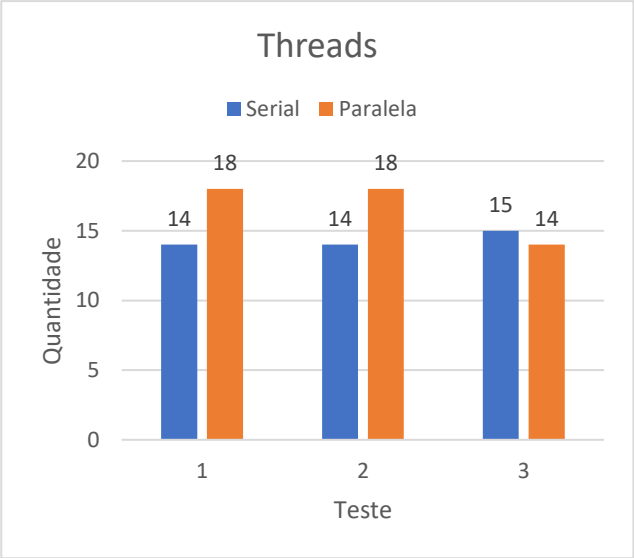


Figura 47: Método 3 – Threads com 100 dados

Os resultados obtidos quanto ao tempo de processamento apresentam no primeiro e segundo teste um gasto de tempo maior por parte da *stream* paralela, que chegou a levar 77,8ms e 46,9ms a mais. No entanto, no terceiro teste houve equivalência entre o tempo gasto no modo serial e paralelo.

Já o uso da CPU, a *stream* serial bateu o maior pico em todos os testes, com uma diferença de 12,3%, 1,5% e 3,8%.

No terceiro cenário, a *stream* paralela teve o maior consumo de memória em todas as ocasiões de testes, sendo a maior dissemelhança de 4Mb.

Quanto ao panorama das *threads*, a *stream* paralela saiu à frente nos dois primeiros testes com a utilização de 4 *threads* a mais em ambos. Contudo, no último teste a situação se inverteu, pois a *stream* serial utilizou 15 *threads*, já a paralela, 14.

QUANTIDADE DE DADOS: 1.000

Dentro do contexto de tempo de processamento paralelo, este se mostrou mais eficiente em todos os cenários, sendo a maior diferença no segundo teste, com 77,3ms a menos que a serial.

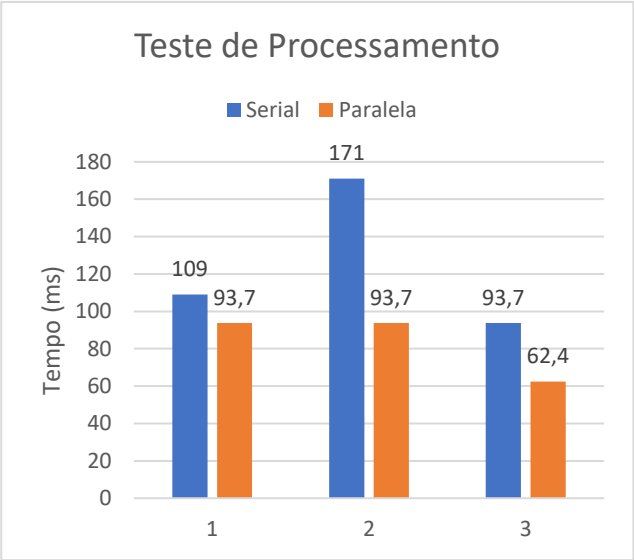


Figura 48: Método 3 – Teste de processamento com 1.000 dados

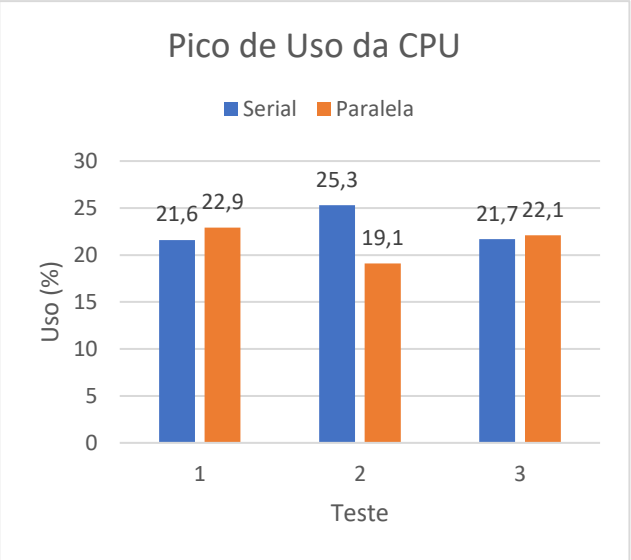


Figura 49: Método 3 – Pico de uso da CPU com 1.000 dados

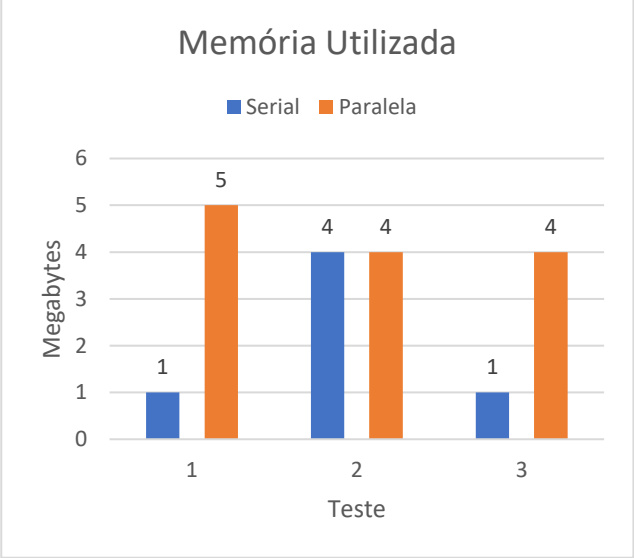


Figura 50: Método 3 – Memória utilizada com 1.000 dados

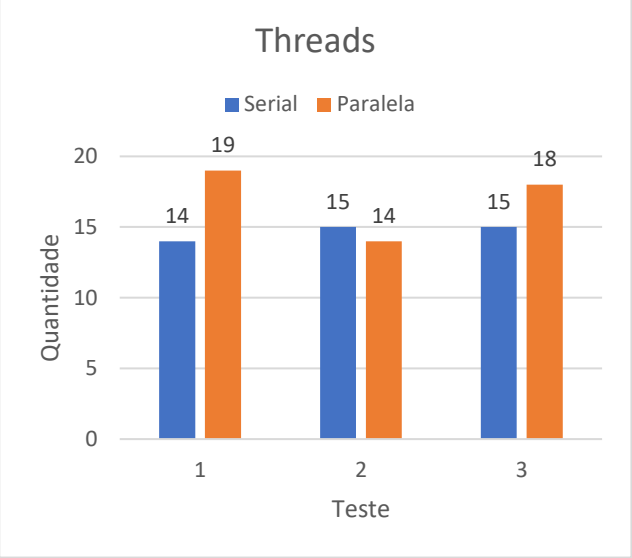


Figura 51: Método 3 – Threads com 1.000 dados

Ademais, o maior pico de uso da CPU ocorreu durante o segundo teste por parte da *stream* serial, com 6,2% de diferença. Embora nos outros testes o modo paralelo que bateu o pico mais alto, a diferença entre eles foi menor, sendo apenas de 1,3% no primeiro teste e 0,4% no terceiro.

Na conjuntura da utilização da memória, aquela que demonstrou ser mais econômica foi a serial com apenas 1Mb de uso no primeiro e terceiro teste, enquanto no segundo houve um empate de 4Mb.

Assim como no uso da memória, a *stream* serial fez menos uso de *threads* no primeiro e último teste. No entanto, no segundo chegou a consumir uma *thread* a mais que a serial. Contudo a maior diferença e utilização aconteceu no primeiro teste por parte da *stream* paralela que fez uso de 19 *threads*.

QUANTIDADE DE DADOS: 10.000

Os resultados obtidos a partir do processamento desta quantia de dados mostram que o tempo gasto pela *stream* serial foi evidentemente maior que a paralela, sendo esta diferença de aproximadamente 6,8x no primeiro teste, 5,6x no segundo e 4,8x no terceiro.

O uso da CPU se mostrou menos custoso por parte da *stream* serial, cujo pico máximo foi de 20,6%. Além disso, este pico permaneceu abaixo do pico mais baixo da paralela que foi de 21,3%, enquanto o mais elevado foi de 23,2%.

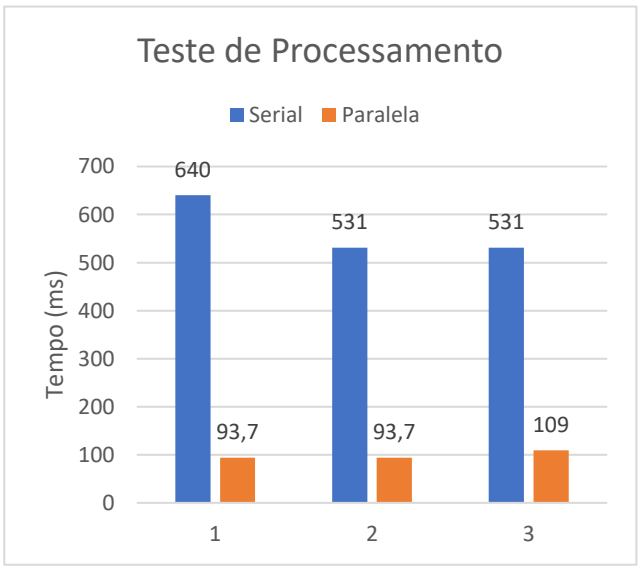


Figura 52: Método 3 – Teste de processamento com 10.000 dados

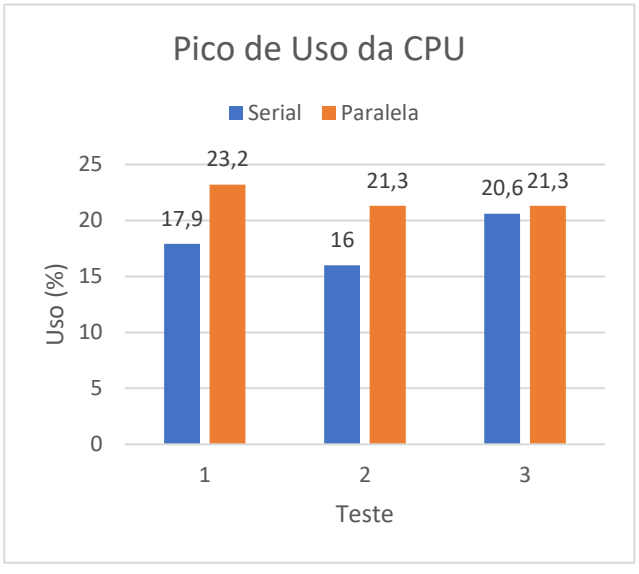


Figura 53: Método 3 – Pico de uso da CPU com 10.000 dados

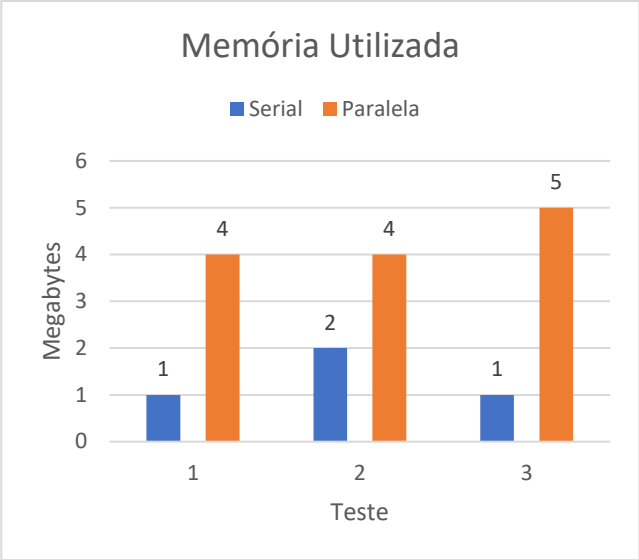


Figura 54: Método 3 – Memória utilizada com 10.000 dados

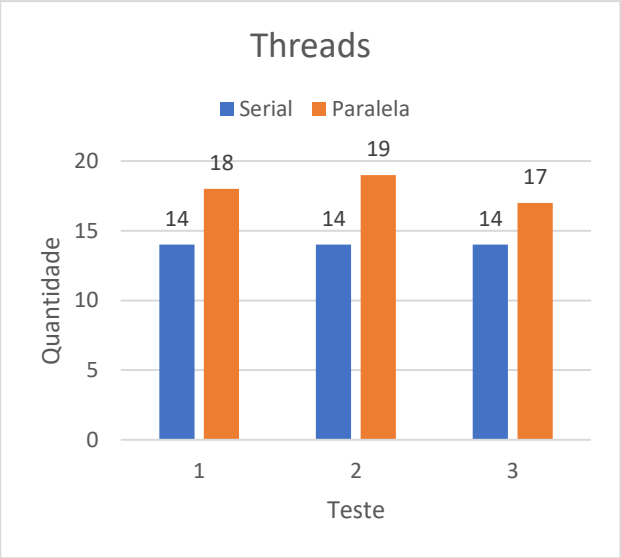


Figura 55: Método 3 – Threads com 10.000 dados

Na análise da memória utilizada a *stream* serial se mostrou mais econômica em comparação à paralela, pois o uso máximo da primeira foi de apenas 2Mb. Por outro lado, o paralelismo fez uso de 4Mb nos dois primeiros testes e 5Mb no último.

Observado a quantidade de *threads* durante a execução do método, concluiu-se que houve uma utilização maior por parte da *stream* paralela com um acréscimo de 4, 5 e 3 em relação à serial.

QUANTIDADE DE DADOS: 100.000

Finalizado os testes com esta base de dados, constatou-se que o tempo de processamento da *stream* serial foi muito superior à paralela, com uma margem mínima de 4.703ms e máxima de 5.063ms.

Quanto ao pico do uso da CPU, o cenário se apresentou desfavorável a *stream* serial, que ficou acima da paralela em todos os testes. A maior diferença ocorreu no terceiro teste com 8,2% de uso a mais.

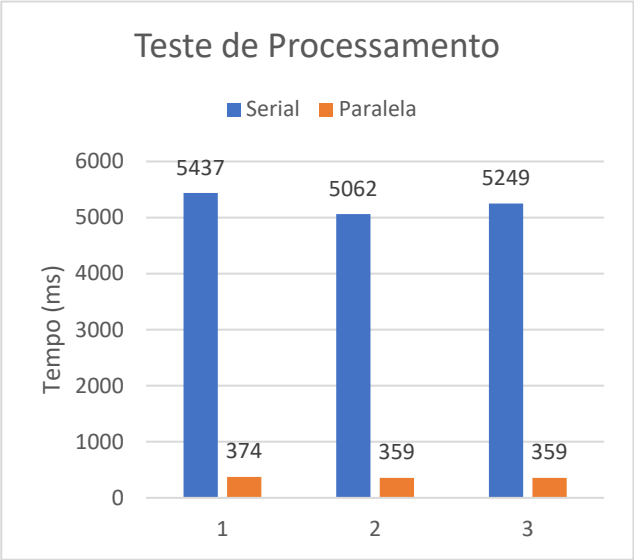


Figura 56: Método 3 – Teste de processamento com 100.000 dados

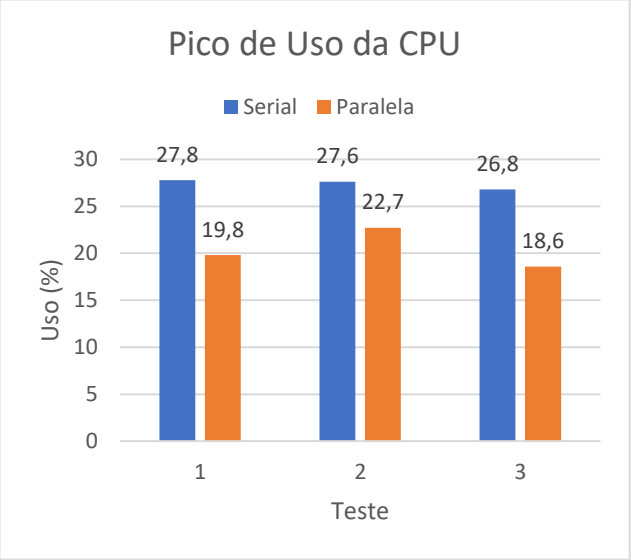


Figura 57: Método 3 – Pico de uso da CPU com 100.000 dados

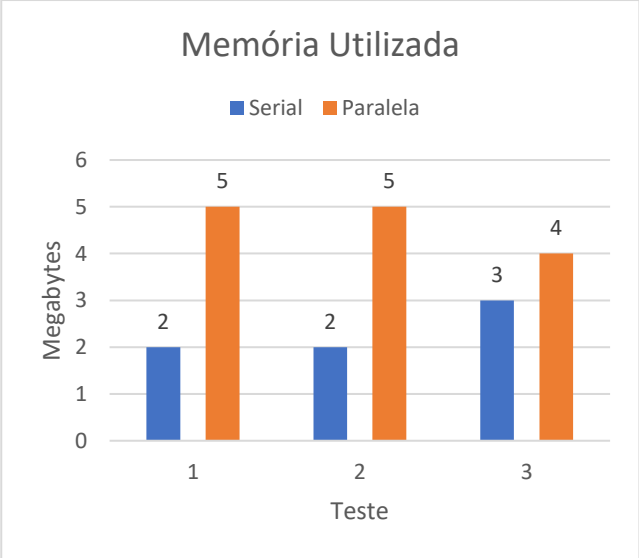


Figura 58: Método 3 – Memória utilizada com 100.000 dados

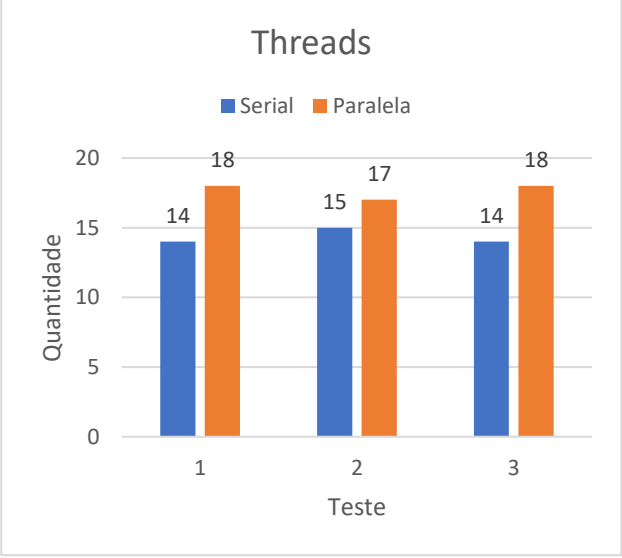


Figura 59: Método 3 – Threads com 100.000 dados

A memória utilizada, de acordo com os resultados, foi por unanimidade maior pela *stream* paralela com até 3Mb a mais utilizados.

O uso de *threads* teve destaque também para a *stream* paralela, que, em comparação a serial, esteve à frente em todos os testes com pelo menos 2 *threads* a mais.

6.3.4. MÉTODO agruparMediaSalarialDePessoasPorPais

QUANTIDADE DE DADOS: 100

De acordo com os dados coletados do tempo de processamento, a *stream* serial teve um custo de tempo menor no segundo e terceiro teste, com uma diferença de 47ms e 31ms, respectivamente. Embora o tempo no primeiro teste tenha sido maior, continua abaixo dos testes subsequentes, visto que a diferença no primeiro foi de apenas 16ms.

A *stream* serial, no contexto do uso da CPU, obteve um pico menor em todos os testes, sendo 2,6%, 1,8% e 1,2% em relação à paralela.

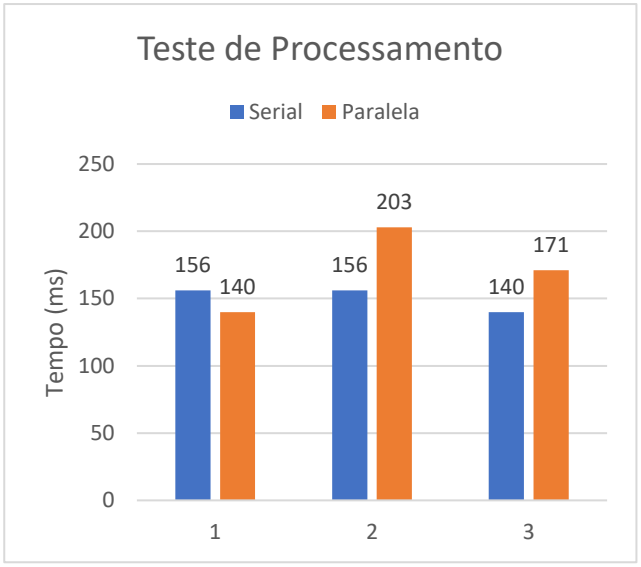


Figura 60: Método 4 – Teste de processamento com 100 dados

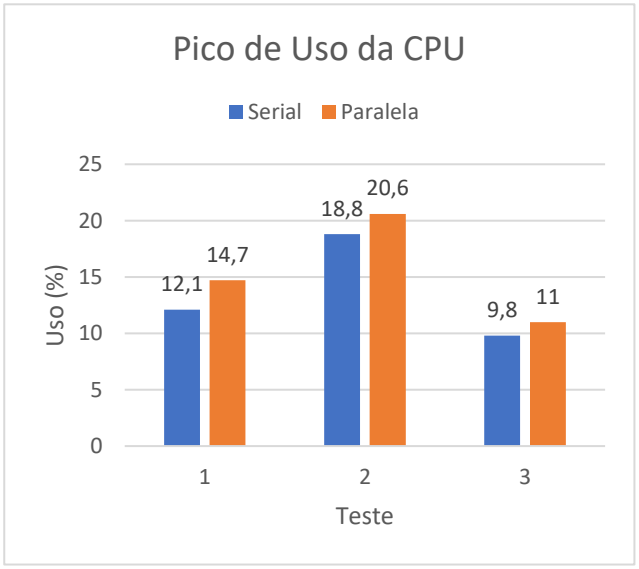


Figura 61: Método 4 – Pico de uso da CPU com 100 dados

O destaque para o uso de memória é da *stream* paralela que ficou acima da serial no primeiro e terceiro teste com 5Mb e 6Mb.O segundo teste ocorreu um empate de 3Mb.

A quantidade de *threads* apresentou resultados que demonstram a maior utilzação por parte do processamento paralelo. Contudo somente o primeiro houve uma inversão por parte da *stream* serial com apenas 1 *thread* a mais.

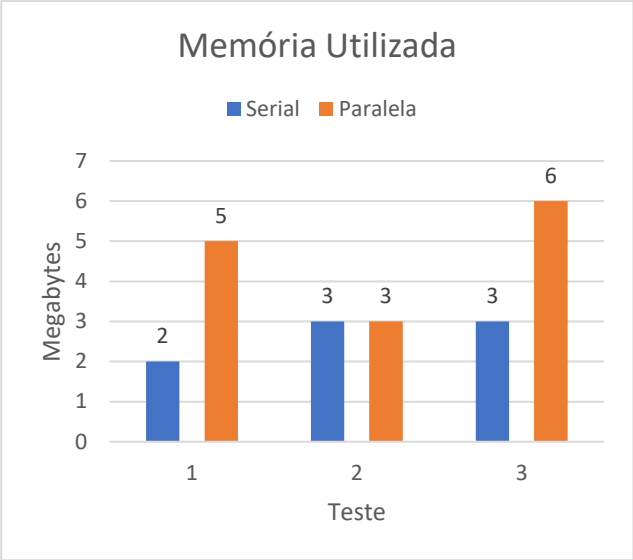


Figura 62: Método 4 – Memória utilizada com 100 dados

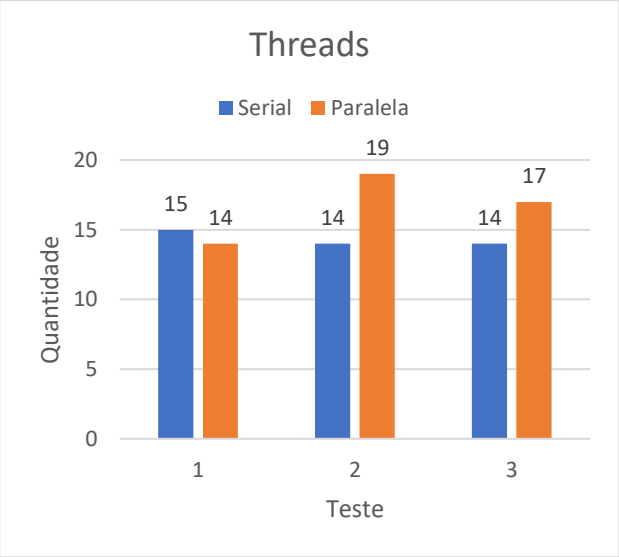


Figura 63: Método 4 – Threads com 100 dados

QUANTIDADE DE DADOS: 1.000

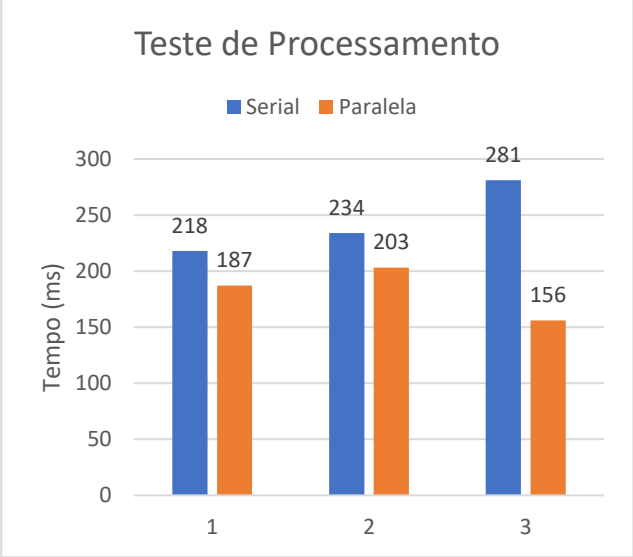


Figura 64: Método 4 – Teste de processamento com 1.000 dados

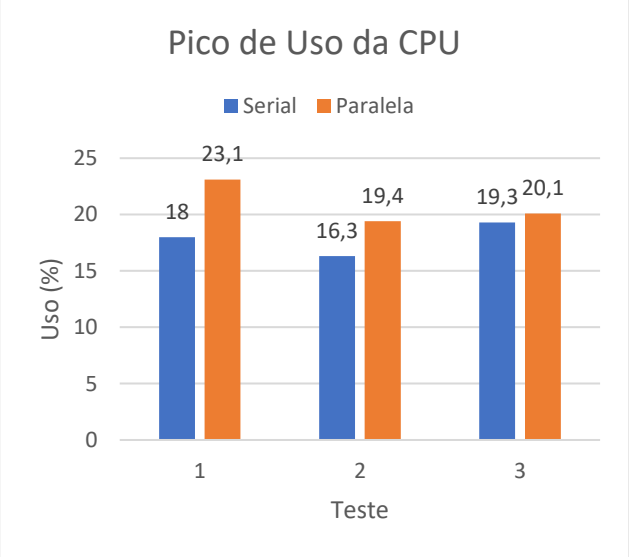


Figura 65: Método 4 – Pico de uso da CPU com 1.000 dados

O tempo de processamento nesta base de dados teve um custo menor por parte da *stream* paralela, com a vantagem de 31ms, 31ms e 125ms. Em contrapartida, o maior pico de uso da CPU ficou por conta da *stream* paralela em todos os testes, com uma margem de diferença de 5,1%, 3,1% e 0,8%, respectivamente.

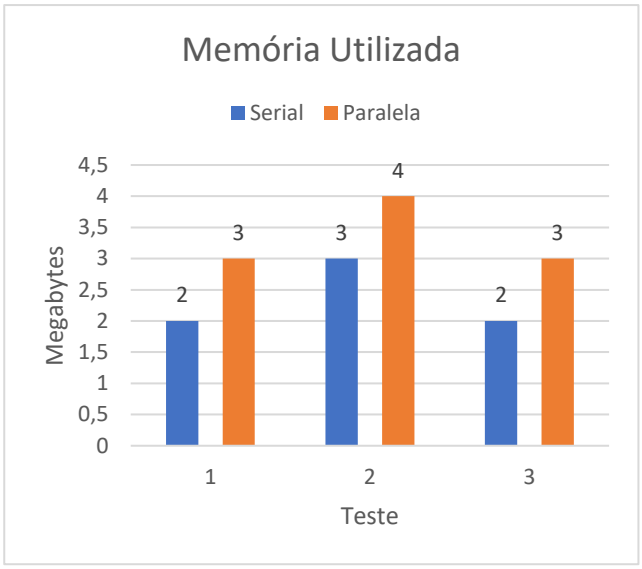


Figura 66: Método 4 – Memória utilizada com 1.000 dados

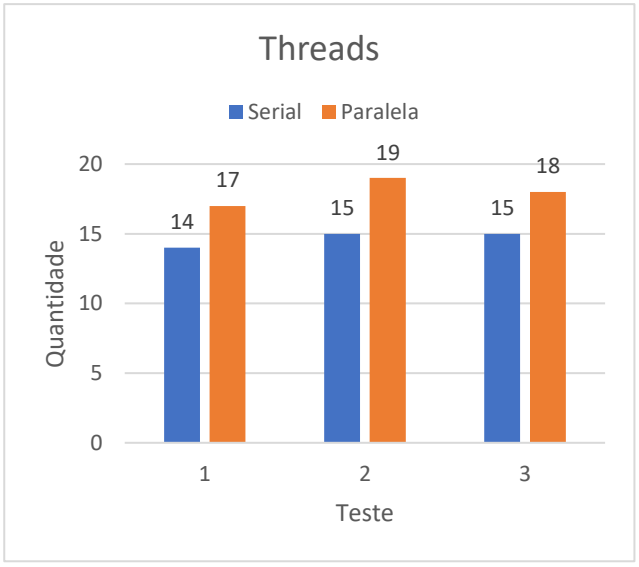


Figura 67: Método 4 – Threads com 1.000 dados

Dentro do cenário desta base de dados, o menor custo de memória (1Mb de diferença em todos os testes) e menor utilização de *threads* (3, 4 e 3 a menos), teve destaque a *stream* serial.

QUANTIDADE DE DADOS: 10.000

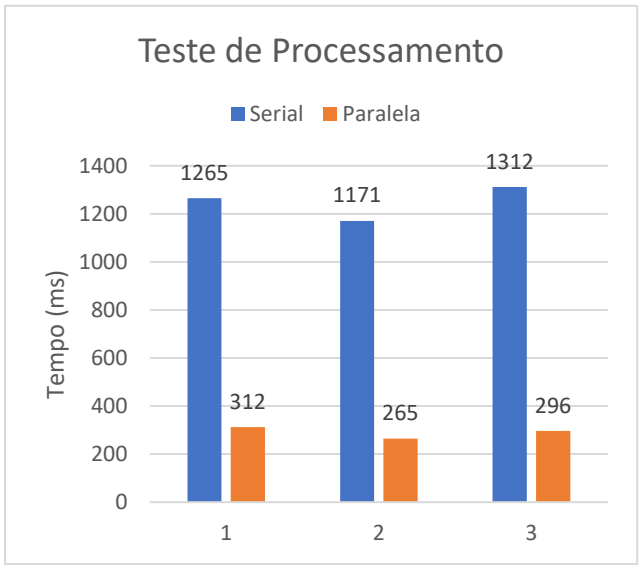


Figura 68: Método 4 – Teste de processamento com 10.000 dados

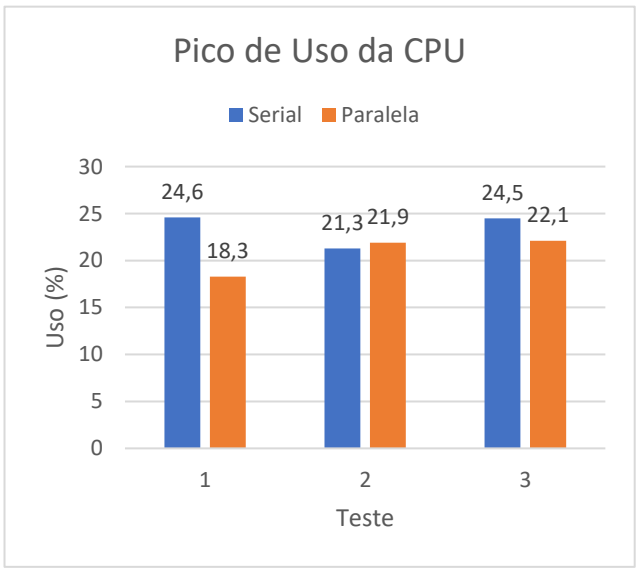


Figura 69: Método 4 – Pico de uso da CPU com 10.000 dados

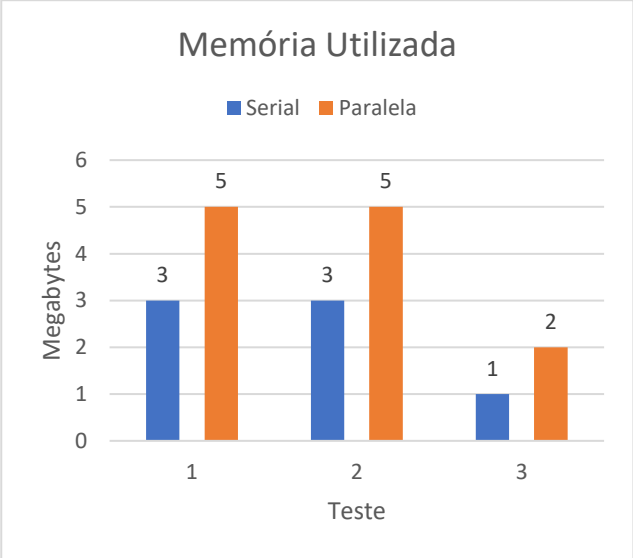


Figura 70: Método 4 – Memória utilizada com 10.000 dados

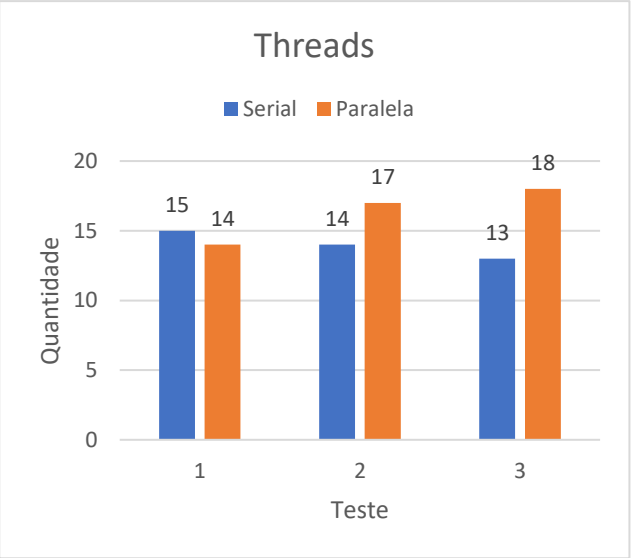


Figura 71: Método 4 – Threads com 10.000 dados

No contexto do tempo de processamento, a maior desvantagem se deu pela *stream* serial, onde o seu menor tempo (1.171ms) ainda foi 906ms a mais que a paralela.

No que tange ao pico de uso da CPU, a *stream* serial também teve destaque no primeiro e último teste, abaixo apenas no segundo teste.

Referente a utilização de memória o realce se dá com a *stream* paralela que está acima em todos os testes com a diferença máxima de 2Mb.

No último contexto de análise a maior quantidade de *threads* usufruídas fica por conta da *stream* paralela no segundo e terceiro teste (3 e 5 a mais que a serial). Atrás apenas no primeiro teste com 1 *thread* a menos.

QUANTIDADE DE DADOS: 100.000

Nesta base de dados processada, o tempo gasto pela *stream* serial foi muito desproporcional à paralela, com duração mínima de 7.796ms (dissemelhança de 6.859ms). A maior discrepância aconteceu no segundo teste com 6.985ms.

Quanto ao pico da CPU, o destaque fica para o primeiro teste, onde a *stream* serial alcançou o pico de 34,7% (13,3% a mais que a paralela). Embora os testes nos testes subsequentes o maior pico é da paralela, as diferenças são de somente 3,2% e 3,1%.

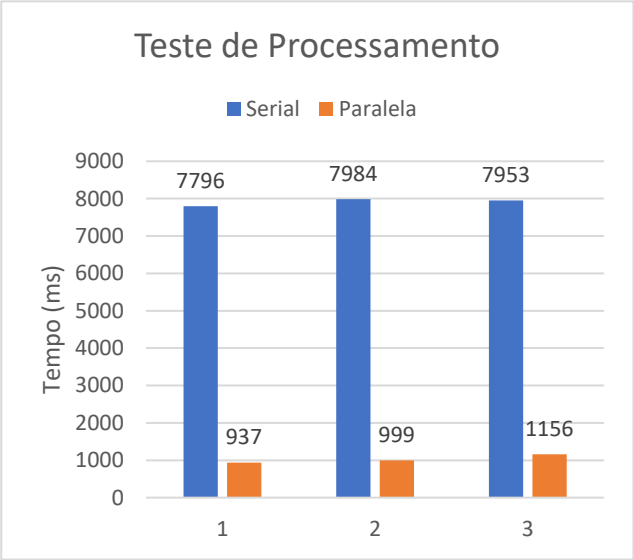


Figura 72: Método 4 – Teste de processamento com 100.000 dados

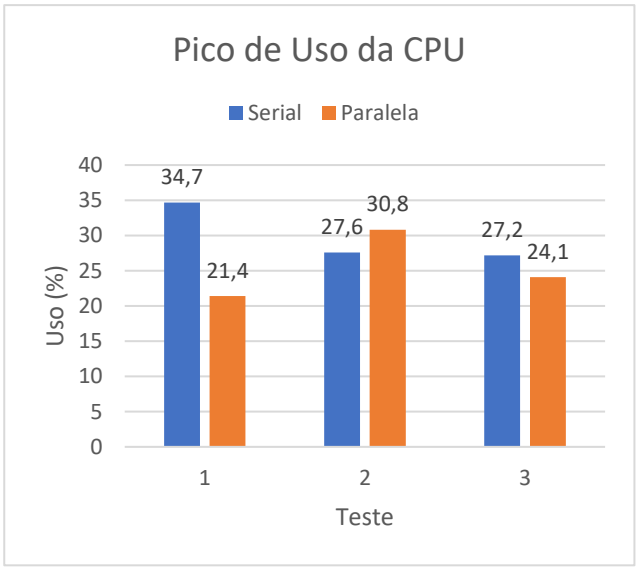


Figura 73: Método 4 – Pico de uso da CPU com 100.000 dados

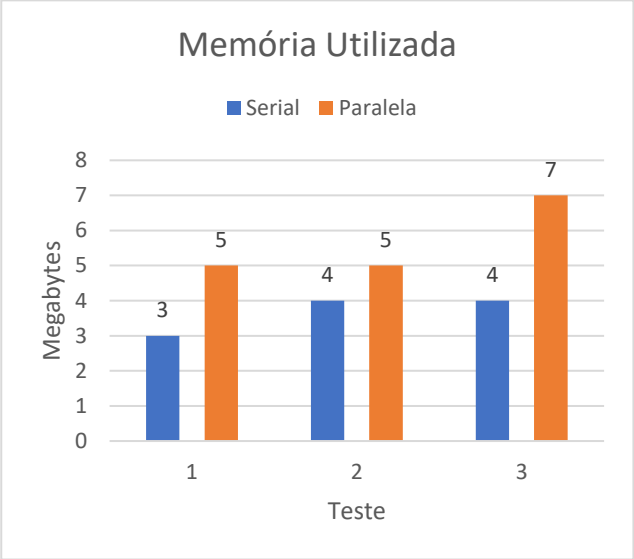


Figura 74: Método 4 – Memória utilizada com 100.000 dados

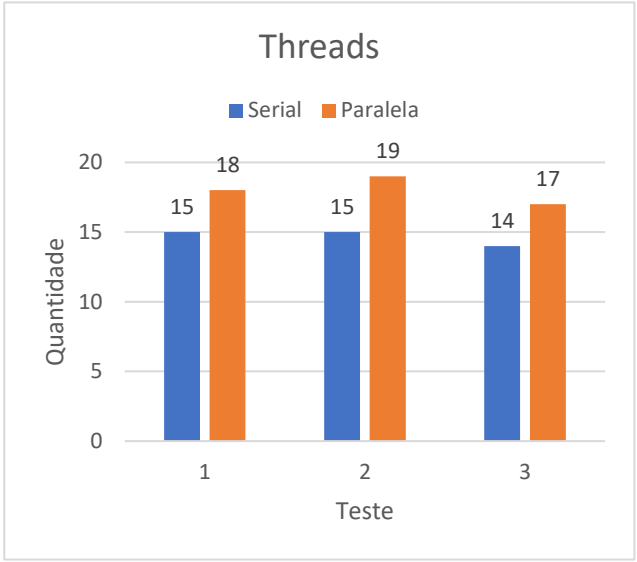


Figura 75: Método 4 – Threads com 100.000 dados

No penúltimo contexto de análise a *stream* paralela saiu à frente em todos os testes, sendo que o terceiro teste obteve o maior realce com o uso de 3Mb a mais de memória. Já no último contexto a diferença oscilou entre 3 e 4 *threads* a mais.

7. CONCLUSÃO

A API de *Stream*, disponibilizada a partir do Java 8 em 2014, trouxe muitos benefícios para a linguagem juntamente com a chegada do paradigma funcional, tais como a redução de código, código mais limpo e flexível, implementações por uma nova perspectiva devido ao paradigma declarativo, onde o programador passa a se preocupar sobre “o que” ele quer fazer ou invés de “como” fazer, além de outras funcionalidade e melhorias na JVM.

A partir desta novidade da programação funcional e as suas vertentes de processamento, sendo elas a serial e paralela, a proposta deste projeto se fundamentou na exploração destes dois modos de trabalhar com dados a fim de obter resultados sobre como seria o seu comportamento e desempenho dentro de cada contexto implementado, levando em consideração a complexidade e a demanda que seria requisitada pelo sistema para executar as operações presentes no *pipeline*. Os resultados gerados pelos testes a serem analisados envolveram o tempo de processamento, o pico de uso da CPU, a utilização de memória e a quantidade de *threads*.

Dito isso, a partir da análise entre os resultados por quantidade de dados dos testes do método buscarMaiorSalarioDePessoasJuridicasCasadasPorPais, nota-se que a média de tempo de processamento foi inferior por parte da *stream* paralela. Este tempo, a princípio, obteve uma diferença ínfima de menos de 6ms, porém no decorrer dos testes com o aumento dos dados para serem processados, esta discrepância aumentava, ao ponto de o tempo ser aproximadamente 6.398ms menor. Já no contexto da média de pico de uso da CPU houve variação conforme a quantidade de dados, sendo a menor média por parte da *stream* serial com 19,63% (2,6% a menos que a paralela) com 100 dados processados e a maior durante os testes com 100.000 dados por parte da *stream* paralela com 36,27% (9,64% a mais que a serial). Quanto a utilização de memória, o maior consumo se deu pela *stream* paralela em todos os testes, com no máximo 4Mb de diferença com 100 e 10.000 dados. No cenário do uso de *threads* a maior utilização também foi por parte da *stream* paralela, que variou entre 16 e 19 *threads*, enquanto a serial teve variação entre 14 e 17, sendo sempre menor em comparação à paralela. Embora houve o maior consumo de memória e *threads* em todas as ocasiões por parte da *stream* paralela, bem como o maior pico e diferença de uso da CPU em comparação à serial, a diferença de tempo gasta durante o processamento faz com que a *stream* paralela tenha uma grande vantagem sobre

a serial, sobretudo em base de dados maiores, onde esta diferença passar a ser cada vez mais perceptível.

Com base nos resultados analisados do método filtrarPessoasPeloSalario constatou-se que a média do tempo de processamento entre as quantidades de dados processadas foi maior em todos os cenários pela *stream* serial. Embora a dissemelhança com 100 dados processados tenha sido apenas 5,1ms, as quantidades subsequentes apresentaram uma diferença maior, sendo 216,93ms, 2.289ms e 6.132ms, respectivamente. No contexto da média de pico de uso da CPU, a *stream* serial saiu à frente em praticamente todas as quantidades de dados, ficando atrás apenas no processamento de 100.000 dados, com uma margem de apenas 0,7%. Já na utilização da memória, a *stream* paralela venceu em praticamente todos os testes, com exceção de um empate de 2Mb durante o último teste com 1.000 dados, enquanto o maior uso de memória foi de 7Mb. Quanto a utilização de *threads*, a *stream* paralela obteve os maiores resultados, exceto pelo segundo teste do processamento de 10.000 dados que ocorreu um empate de 16 *threads*. Apesar dos resultados não serem muito favoráveis a *stream* paralela quanto ao uso de *threads* e memória, também não favoreceu muito a serial quanto ao pico de uso da CPU. No entanto, a principal e maior diferença nos resultados deste método se dá pelo tempo de processamento, cuja a execução paralela obteve uma vantagem cada vez maior conforme a quantidade de dados aumentava.

Encerrado os testes do método buscarPessoaQuePossuiMenorSalario, nota-se que a média de tempo gasta para processamento pela *stream* paralela com 100 dados foi praticamente o dobro da *stream* serial, porém nas quantidades posteriores o cenário se inverte e quanto maior a base de dados, melhor a performance da paralela. De modo geral, a média do pico da CPU foi menor somente com 10.000 dados por parte da *stream* serial, já o seu maior pico ocorreu durante o processamento com 100.000 dados que atingiu a média de 27,40%, 7,03% a mais que a paralela. Já no contexto de uso de memória, a *stream* paralela esteve à frente em praticamente todos os testes, com variação de utilização entre 3Mb e 5Mb. A única exceção aconteceu durante o processamento de 1.000 dados no segundo teste, com um empate de 4Mb. Quanto ao cenário das *threads* a *stream* paralela esteve atrás somente no terceiro teste com o processamento de 100 dados e uma diferença ínfima de 1 *thread*. A serial manteve um uso que variou entre 14 e 15, enquanto a paralela entre 14 e 19. Dito isso, a maior discrepância ocorreu na análise do tempo de

processamento, que apesar dos outros contextos analisados, a *stream* serial mantém a vantagem no processamento de 100 dados e a paralela nas quantidades subsequentes.

A partir dos resultados do método agruparMediaSalarialDePessoasPorPais, observou-se que a média de tempo que levou para processá-lo foi menor por parte da *stream* serial somente com 100 dados, onde obteve uma vantagem de 20,66ms. Nos demais testes a *stream* paralela passou a ter uma performance cada vez melhor, com um custo de tempo de aproximadamente 7,6 vezes menor com em relação a serial no processamento de 100.000 dados. Além disso, a média de pico de uso da CPU foi maior por parte da *stream* paralela com 100 e 1.000 dados processados; já a serial atingiu o maior pico nas bases de 10.000 e 100.000 dados. Quanto ao uso de memória, a *stream* paralela obteve os maiores valores em quase todos os testes, com exceção apenas no segundo testes com 100 dados, onde ocorreu um empate de 3Mb, enquanto nos demais, o uso variou entre 2Mb e 7Mb. Por último, a utilização das *threads* também teve destaque para a *stream* paralela, que neste contexto ficou com o menor uso em apenas dois testes (o primeiro teste com 100 e 10.000 dados). Conquanto a *stream* paralela tenha feito uso de mais recursos da CPU, memória e *threads*, ao final esta demanda se mostrou favorável, pois ao analisar o tempo conforme o aumento da quantidade de dados para serem processados há uma redução cada vez mais perceptível no tempo em relação a serial.

7.1. TRABALHOS FUTUROS

Com base no tema proposto, pode-se dar continuidade à exploração e análise de novos resultados a partir dos métodos já existentes e também de novas implementações por meio da API de *Stream*, a fim de examinar, desta vez, qual será o comportamento e desempenho destes mesmos métodos com o uso de diferentes versões da JDK posteriores ao Java 8. Além disso, outra vertente a ser explorada seria a aplicação desta API dentro do contexto de gerenciamento filas, sobretudo em aplicações distribuídas e quais os impactos que a forma de utilização de *Stream* poderia ter, bem como os efeitos do uso síncrono e assíncrono dentro deste contexto.

REFERÊNCIAS

AMORIM, G. **Java Performance: Aprimorando o desempenho de aplicações**. 2014. Disponível em: <https://www.devmedia.com.br/java-performance-aprimorando-o-desempenho-de-aplicacoes/31277>. Acesso em 31 out. 2021.

AMORIM, G. **Streams API: trabalhando com coleções de forma flexível em Java**. 2015. Disponível em: <https://www.devmedia.com.br/streams-api-trabalhando-com-colecoes-deforma-flexivel-em-java/31980>. Acesso em 18 out. 2021.

ATENCIO, L. **Understanding Lambda Expressions**. 2015. Disponível em: <https://medium.com/@luijar/understanding-lambda-expressions-4fb7ed216bc5>. Acesso em 27 dez. 2021.

CARVALHO, M. **Java 8 – Functional Interfaces – Tornando o Java mais legal**. 2018. Disponível em: <https://medium.com/@mvalho/java-8-functional-interfaces-tornando-o-java-mais-legal-72401462d0e2>. Acesso em 15 mar. 2022.

COSTA, S. **Uma visão muito breve sobre o paradigma funcional**. Disponível em: <https://medium.com/@sergiocosta/paradigma-funcional-3194924a8d20>. Acesso em 12 dez. 2021.

DELGADO, J.; RIBEIRO, C. **Arquitetura de Computadores**. 5ª ed. Tradução de Elvira Maria Antunes Uchôa. Rio de Janeiro: LTC, 2017.

DIONISIO, E. **Introdução ao Java JDK**. 2013. Disponível em: <https://www.devmedia.com.br/introducao-ao-java-jdk/28896>. Acesso em 22 fev. 2022.

FACHOLI, R. **Monitorando a sua aplicação Java com VisualVM**. 2015. Disponível em: <https://www.redspark.io/monitorando-sua-aplicacao-java-com-visualvm/>. Acesso em 02 jul. 2022.

FURGERI, S. **Java 8 – Ensino Didático: Desenvolvimento e implementação de aplicações**. São Paulo: Érica, 2015.

HENNESSY, J.; PATTERSON, D. **Arquitetura de computadores: Uma abordagem quantitativa**. 6ª ed. Tradução de Daniel Vieira. Rio de Janeiro: Elsevier, 2019.

INFORCHANNEL. **Oracle apresenta o Java 18 com diversas novidades para os desenvolvedores**. 2022. Disponível em: <https://inforchannel.com.br/2022/03/22/oracle-apresenta-o-java-18-com-diversas-novidades-para-os-desenvolvedores/>. Acesso em 31 jul. 2022.

KEOMA, D. **As vantagens da Programação Funcional (PF)**. 2018. Disponível em: <https://administradores.com.br/artigos/as-vantagens-da-programacao-funcional-pf>. Acesso em 27 dez. 2021.

KRILL, P. **Java 19 could be big**. 2022. Disponível em: <https://www.infoworld.com/article/3652336/java-19-may-be-quite-ambitious.amp.html>. Acesso em 31 jul. 2022.

MACHADO, F.; MAIA, L. **Arquitetura de Sistemas Operacionais**. 5ª ed. Rio de Janeiro: LTC, 2017.

MEDEIROS, H. **Programação com Threads**. 2007. Disponível em: <https://www.devmedia.com.br/programacao-com-threads/6152>. Acesso em 31 dez. 2021.

MELO, A.; SILVA, F. **Princípios de linguagens de programação**. 3ª ed. São Paulo: Edgard Blücher, 2014.

MENDES, L. **O que é, e para que serve o Cronograma**. 2015. Disponível em: <https://docplayer.com.br/5464728-O-que-e-e-para-que-serve-o-cronograma.html>. Acesso em 05 out. 2021.

OTTERO, R. **Introdução ao Maven**. Disponível em: <https://www.devmedia.com.br/introducao-ao-maven/25128>. Acesso em: 15 mar. 2022.

PEREIRA, R. **Programação funcional com Java**. 2015. Disponível em: <https://www.devmedia.com.br/programacao-funcional-com-java/32176>. Acesso em 14 dez. 2021.

REZENDE, J. **Começando com Programação Funcional**. 2018. Disponível em: <https://medium.com/trainingcenter/começando-com-programação-funcional-de389de2b8fe>. Acesso em 16 dez. 2021.

SAHU, K. **A use case with Python Faker Library**. 2021. Disponível em: <https://medium.com/plumbersofdatascience/a-use-case-with-python-faker-library-77d343af3f09>. Acesso em: 16 mar. 2022.

SAUMONT, P. **Functional Programming in Java: How functional techniques improve your Java programs**. New York: Manning, 2017.

SCHILDT, H. **Java para iniciantes: crie, compile e execute programas java rapidamente**. 6ª ed. Tradução de Aldir José Coelho Corrêa da Silva. Porto Alegre: Bookman, 2015

SEBESTA, R. **Conceitos de linguagens de programação**. 11ª ed. Tradução de João Eduardo Nóbrega Tortello. Porto Alegre: Bookman, 2018.

SILVA, C. **Java 8: Iniciando o desenvolvimento com a Streams API**. 2016. Disponível em: <https://www.oracle.com/br/technical-resources/articles/java-stream-api.html>. Acesso em: 18 out. 2021.

SILVA, C. **Java 8: Iniciando o desenvolvimento com a Streams API**. 2017. Disponível em: <https://www.infoq.com/br/articles/java8-iniciando-desenvolvimento-com-a-streams-api/>. Acesso em: 09 mar. 2022.

SILVA, C. **Java Streams API: manipulando coleções de forma eficiente**. 2017. Disponível em: <https://www.devmedia.com.br/java-streams-api-manipulando-colecoes-de-forma-eficiente/37630>. Acesso em: 08 mar. 2022.

SILVA, F.; LEITE, M.; OLIVEIRA, D. **Paradigmas de programação**. Porto Alegre: SAGAH, 2019.

SILVA, N. **Teste de Performance e a sua importância**. 2019. Disponível em: < https://medium.com/@nikollasfs22/teste-de-performance-e-a-sua-importância-2a4271fe37a6>. Acesso em: 17 out. 2021.

SILVEIRA, P.; TURINI, R. **Java 8 Prático: Lambdas, Streams e os novos recursos da linguagem**. Casa do Código.

SILVEIRA, S.; et al. **Paradigmas de programação: uma introdução**. Belo Horizonte: SYNAPSE, 2021.

SOUZA, D. **Uma visão sobre o Projeto Lombok**. 2013. Disponível em: < https://www.devmedia.com.br/uma-visao-sobre-o-projeto-lombok/28321>. Acesso em 16 mar. 2022.

SUBRAMANIAM, V. **Function Programming in Java: Harnessing the Power of Java 8 Lambda Expressions**. Texas: The Pragmatic Bookshelf. 2014.

SUBRAMANIAM, V. **Interfaces funcionais: Saiba como criar interfaces funcionais customizadas e entenda por que você deve usar integrações sempre que possível**. 2017. Disponível em: <https://developer.ibm.com/br/articles/j-java8idioms7/>. Acesso em: 15 mar. 2022.

TEDESCO, K. **Concorrência, Paralelismo, Processos, Threads, programação síncrona e assíncrona**. 2020. Disponível em: <https://www.treinaweb.com.br/blog/concorrencia-paralelismo-processos-threads-programacao-sincrona-e-assincrona>. Acesso em 31 dez. 2021.

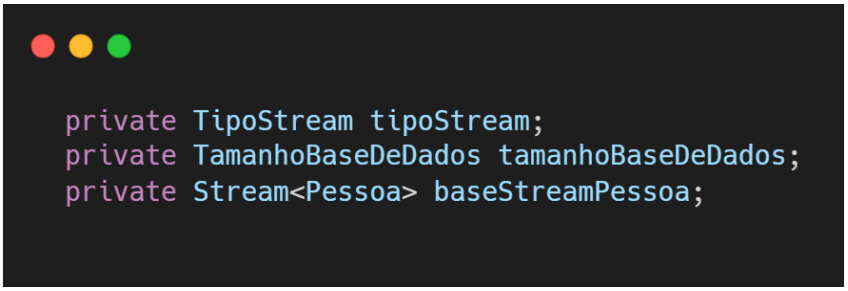
TUCKER, A.; NOONAN R. **Linguagens de Programação: Princípios e Paradigmas**. 2ª ed. Tradução de Mario Moro Fecchio e Acauan Fernandes. São Paulo: AMGH. 2010.

URMA, R.; FUSCO M.; MYCROFT, A. **Modern Java in Action: Lambdas, streams, functional and reactive programming**. New York: Manning, 2019.

ZANIN, A.; et al. **Qualidade de software**. Porto Alegre: SAGAH, 2018.

APÊNDICE A – Classe TesteStreamPessoa

Esta classe contém 3 atributos privados: o primeiro, responsável por determinar qual o tipo de *Stream* que processará os dados; o segundo, para estabelecer o tamanho da base de dados; e por último uma *Stream* de Pessoa que conterá os dados das pessoas e a forma de processamento definida previamente de acordo com os dois atributos mencionados anteriormente.



```
private TipoStream tipoStream;
private TamanhoBaseDeDados tamanhoBaseDeDados;
private Stream<Pessoa> baseStreamPessoa;
```

Figura 76: Atributos da classe TesteStreamPessoa

Dito isso, os atributos referentes ao tipo da *Stream* e o tamanho da base de dados são recebidos via construtor no momento da instanciação.

```
public TesteStreamPessoa(TipoStream tipoStream, TamanhoBaseDeDados tamanhoBaseDeDados) {
    this.tipoStream = tipoStream;
    this.tamanhoBaseDeDados = tamanhoBaseDeDados;
}
```

Figura 77: Construtor da classe TesteStreamPessoa

Após a instanciação do objeto, se faz necessário configurá-lo conforme os parâmetros recebidos via construtor. A configuração irá determinar os detalhes da forma de processamento do método para teste e a quantidade de dados a ser processada.

```java
public void configurar() {
    if (tipoStream.equals(TipoStream.SERIAL))  {
        baseStreamPessoa = PessoaRepository.getPessoas()
                .stream()
                .limit(tamanhoBaseDeDados.getQuantidade());
    }
    else {
        baseStreamPessoa = PessoaRepository.getPessoas()
                .parallelStream()
                .limit(tamanhoBaseDeDados.getQuantidade());
    }

    PessoaRepository.limparLista();
}
```

Figura 78: Método para configuração da classe TesteStreamPessoa

```java
TipoStream tipoStream = TipoStream.PARALELA;
TamanhoBaseDeDados tamanhoBaseDeDados = TamanhoBaseDeDados.CEM_MIL;

TesteStreamPessoa testeStreamPessoa = new TesteStreamPessoa(tipoStream, tamanhoBaseDeDados);
testeStreamPessoa.configurar();
```

Figura 79: Instanciação e configuração da classe TesteStreamPessoa