INSTITUTO FEDERAL DE MINAS GERAIS
CAMPUS SÃO JOÃO EVANGELISTA

HERON LIMA DE OLIVEIRA; VÍCTOR ALVES FERREIRA CUNHA

UM ESTUDO EXPLORATÓRIO SOBRE PROGRAMAÇÃO PARALELA UTILIZANDO JAVASCRIPT

SÃO JOÃO EVANGELISTA
2021

HERON LIMA DE OLIVEIRA; VÍCTOR ALVES FERREIRA CUNHA

UM ESTUDO EXPLORATÓRIO SOBRE PROGRAMAÇÃO PARALELA UTILIZANDO JAVASCRIPT

Trabalho de conclusão de curso apresentado ao Instituto Federal de Minas Gerais - *Campus* São João Evangelista como exigência parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Me. Rosinei Soares de Figueiredo

SÃO JOÃO EVANGELISTA

2021

REDE DE BIBLIOTECAS

FICHA CATALOGRÁFICA PARA TRABALHO DE CONCLUSÃO DE CURSO

C972u Cunha, Victor Alves Ferreira; Oliveira, Heron Lima de.

Um estudo exploratório sobre programação paralela utilizando JavaScript [manuscrito] / Victor Alves Ferreira Cunha; Heron Lima de Oliveira. – 2021.

41f.: il.

Orientador: Me. Rosinei Soares de Figueiredo.

Trabalho de Conclusão de Curso (bacharelado) – Instituto Federal Minas Gerais. *Campus* São João Evangelista, 2021.

1. Programação. 2. JavaScript. 3. Paralelismo. 4. Web Workers. I. Cunha, Victor Alves Ferreira. II. Oliveira, Heron Lima de. III. Instituto Federal de Minas Gerais. *Campus* SJE. IV. Título.

CDD 005.275

Catalogação: Rejane Valéria Santos - CRB-6/2907

HERON LIMA DE OLIVEIRA; VÍCTOR ALVES FERREIRA CUNHA

UM ESTUDO EXPLORATÓRIO SOBRE PROGRAMAÇÃO PARALELA UTILIZANDO JAVASCRIPT

Trabalho de conclusão de curso apresentado ao Instituto Federal de Minas Gerais - *Campus* São João Evangelista como exigência parcial para obtenção do título de Bacharel em Sistemas de Informação.

Aprovado em 12/04/2021

BANCA EXAMINADORA

Orientador: Prof. Me. Rosinei Soares de Figueiredo
Instituto Federal de Minas Gerais – *Campus* São João Evangelista

Convidado: Prof. Me. Fábio Rodrigues Martins
Instituto Federal de Minas Gerais – *Campus* São João Evangelista

Convidado: Prof. Me. André Luyde da Silva Souza
Instituto Federal de Minas Gerais – *Campus* São João Evangelista

SÃO JOÃO EVANGELISTA

2021

RESUMO

É possível afirmar que a implementação de códigos computacionais utilizando programação paralela demanda mais tempo e esforço dos programadores quando comparada às codificações utilizando programação sequencial. Outro fator que difere as duas abordagens é a quantidade de material didático disponibilizado, principalmente se tratando de meios digitais e de conteúdo escrito em língua portuguesa. Quando o objetivo é a utilização de paralelismo em um código *JavaScript* (JS) o cenário torna-se ainda mais desfavorável, visto que a própria linguagem apresenta barreiras que dificultam a implementação da programação paralela. Para solucionar esses "problemas" apresentados pelo JS, surgiu através de um esforço colaborativo entre Mozilla e Google, uma API chamada *Web Workers*. Essa tecnologia permite que *scripts* possam ser executados em outras *threads*, simultaneamente à *thread* principal. Estudos e materiais didáticos a respeito da implementação do paralelismo em JS são ainda mais escassos. Em razão disso, o presente trabalho teve como objetivo contribuir com a difusão da programação paralela e sua utilização em linguagem *JavaScript*. Dentre os procedimentos desenvolvidos, pode-se destacar a elaboração do documento escrito, que visa contribuir de alguma forma com a área de pesquisa; bem como a construção de uma solução computacional, que reforça os argumentos apresentados no trabalho. A metodologia utilizada neste estudo é de caráter qualitativo, visto que a maioria das informações coletadas pelos pesquisadores estavam expressas em forma de texto, sem a presença de números que, de alguma forma, pudessem interferir na elaboração dos argumentos apresentados. Por outro lado, é possível dizer que, após a realização de testes de desempenho nas aplicações desenvolvidas, o trabalho recebeu características de um estudo quantitativo, expressando em números os dados obtidos. Os resultados demonstraram que a paralelização dos códigos proporcionou um ganho expressivo no desempenho da aplicação e, em decorrência disto, proporcionaram também uma melhor experiência ao usuário.

Palavras-chaves: Programação. *JavaScript*. Paralelismo. *Web Workers*.

ABSTRACT

The implementation of computational codes using parallel programming demands more time and effort from programmers when compared to coding using sequential programming. Another factor that differentiates the two approaches is the amount of didactic material available, mainly in the case of digital media and content written in Portuguese. Paralleling a JavaScript code (JS) is even more complicated, since the language itself presents barriers that hinder the implementation of parallel programming. To solve these "problems" presented by JS, it came about through a collaborative effort between Mozilla and Google, an API called Web Workers. This technology allows scripts to be executed on other threads, simultaneously with the main thread. Studies and teaching materials regarding the implementation of parallelism in JS are even more scarce. As a result, this research aimed to contribute to the spread of parallel programming and its use in JavaScript language. Among the procedures developed, it is possible to highlight the creation of the written document, which aims to contribute in some way to the research area. It is also possible to highlight the construction of a computational solution, which reinforces the arguments presented in this project. The methodology used in this study is of a qualitative nature, since most of the information collected by the researchers was expressed in text form, without the presence of numbers that, in some way, could interfere in the elaboration of the arguments presented. After performing performance tests on the developed applications, this research received characteristics of a quantitative study, expressing the data obtained in numbers. The results demonstrated that the parallelization of the codes provided an expressive gain in the performance of the application and a better user experience.

Keywords: Programming. JavaScript. Parallelism. Web Workers.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Devido aos grandes avanços da tecnologia, é possível perceber que há uma crescente demanda humana por ganhos de desempenho durante a utilização de computadores e dispositivos móveis. Além disso, com o passar dos anos, mais complexas tornam-se as operações realizadas pelos programas (*softwares)*. Dessa forma, é crucial que as empresas de *hardware* acompanhem as necessidades requisitadas por esses *softwares* e também pelos usuários.

Quando o assunto é a parte física de um computador, ou de dispositivos móveis, sabe-se que a Unidade Central de Processamento (UCP), do inglês *Central Processing Unit* (CPU), é o componente de *hardware* que possui maior importância no funcionamento do equipamento. "A CPU é responsável por calcular e realizar tarefas determinadas pelo usuário e é considerada o cérebro do PC" (CANALTECH, 2015).

Nos dias atuais, as CPU's, que também podem ser chamadas de processadores, são fabricadas com múltiplos núcleos (*multicores*). Cada núcleo funciona como se fosse um processador independente e, dessa forma, os processos são divididos entre esses diversos núcleos, resultando em um ganho de desempenho no dispositivo utilizado. "Processadores *multicore* oferecem várias vantagens: podem realizar duas ou mais tarefas ao mesmo tempo; um núcleo pode trabalhar com uma velocidade menor que o outro, reduzindo a emissão de calor; ambos podem compartilhar memória cache; entre outros" (ALECRIM, 2008).

De acordo com Cipoli (2012), apesar das frequentes evoluções dos equipamentos de *hardware*, mais especificamente dos processadores, ainda são poucos os *softwares* que conseguem tirar proveito de toda a capacidade de processamento oferecida pelas máquinas. Algumas poucas aplicações, como editores de vídeos, jogos eletrônicos e programas CAD costumam tirar proveito de todas essas CPU's.

Tradicionalmente, os programas computacionais são construídos de maneira sequencial, ou seja, seus comandos são executados um após o outro, na sequência em que foram especificados. Esses programas são executados por um único processador e, suas instruções, que poderiam ser divididas entre os diversos núcleos disponibilizados, acabam sendo executadas somente por um deles, deixando os demais ociosos.

Uma forma de contornar esse problema é a construção de programas e aplicativos utilizando a chamada programação paralela. Diferentemente da programação sequencial, esse tipo de programação tira proveito dos múltiplos *cores* do processador, fazendo com que as

aplicações construídas utilizando essa abordagem decomponham os processos em partes e, em seguida, distribuam todas essas partes entre os diversos núcleos.

Segundo Loewe (2019), com a utilização do paralelismo, conseguimos ainda, aumentar a vida útil dos componentes, reduzir o tempo utilizado na resolução de problemas computacionais e até mesmo diminuir o consumo de energia em alguns casos. Dessa forma, pode-se dizer que ganhos significativos são obtidos tanto pelos usuários quanto pelo meio ambiente.

Loewe (2019) afirma que a programação paralela também oferece a oportunidade de trabalharmos com problemas maiores e mais complexos, pois, devido ao fato de se estar utilizando mais de um núcleo da CPU, haverá uma redução da carga de trabalho individual e um melhor gerenciamento da memória e processador.

O paralelismo pode ser utilizado em diversas linguagens de programação, porém, neste trabalho, utilizamos a linguagem *JavaScript* (JS). De acordo com Zampieri (2019), o JS permite ao desenvolvedor implementar diversos itens de alto nível de complexidade em páginas *web*, como animações, mapas, gráficos ou informações que se atualizam em intervalos de tempo. O JS foi escolhido por ser uma linguagem presente na maioria dos projetos *web* e pelo fato de ser uma linguagem que apresenta algumas barreiras em relação ao uso do paralelismo, isso faz com que um fator motivador e desafiador seja acrescido ao estudo.

Apesar de robusto e oferecer diversas possibilidades, o JS carece de recursos próprios para efetuar a utilização dos múltiplos núcleos e *threads* disponibilizados por um dispositivo. Por se tratar de uma linguagem *single threaded*, o *JavaScript* executa uma única operação por vez, até que a mesma seja concluída. Para contornar essas dificuldades apresentadas pelo JS, tornou-se necessário a utilização de uma *Application programming interface* (API) denominada *WebWorkers*, essa API é posteriormente detalhada neste trabalho.

Diante do supracitado, o presente trabalho teve como objetivo geral contribuir com a difusão da programação paralela e sua utilização com *JavaScript*, visto que é uma abordagem ainda pouco estudada. Para sustentar esta pesquisa, um estudo de caso foi utilizado no desenvolvimento de uma solução computacional que reforce as ideias apresentadas.

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos: a) realizar um estudo sobre a programação paralela e sua utilização na linguagem de programação *JavaScript*; b) selecionar um estudo de caso que possa ser utilizado no desenvolvimento de uma aplicação; c) desenvolver a aplicação de duas formas, uma utilizando a programação sequencial e outra utilizando a programação paralela; d) realizar os devidos testes de desempenho nas aplicações desenvolvidas; e) comparar os resultados obtidos e apresentá-los.

2. REFERENCIAL TEÓRICO

Esse capítulo apresenta informações que fundamentam essa pesquisa, levando em consideração estudos e considerações feitos por diversos autores que são indispensáveis para o prosseguimento da mesma.

2.1. PROCESSADORES E SUA EVOLUÇÃO

Os componentes físicos que se encontram dentro dos dispositivos, sejam computadores ou dispositivos móveis, são denominados *hardwares*. Cada um desses componentes possui extrema importância para o funcionamento do todo. Entretanto, o processador (CPU) merece uma atenção especial, visto que, este componente é responsável por todas as instruções e operações realizadas nos dispositivos.

De acordo com Alecrim (2008), os processadores "são chips responsáveis pela execução de cálculos, decisões lógicas e instruções que resultam em todas as tarefas que um computador pode fazer. Por este motivo, são também referenciados como cérebros destas máquinas".

Os processadores têm uma série de componentes internos e características que descrevem sobre seu funcionamento, porém, para um melhor entendimento do assunto tratado neste trabalho, apenas algumas dessas características necessitam ser compreendidas. No capítulo a seguir, é feita uma síntese sobre as funções exercidas pelo núcleo de um processador e como funcionam os múltiplos núcleos.

2.1.1. Núcleos e Multinúcleos

As primeiras CPU's eram fabricadas contendo somente um núcleo (*single core*). Com a chegada de novas tecnologias e a necessidade de um maior poder de processamento, as empresas fabricantes de processadores começaram a produzir CPU's que continham mais de um núcleo (*multi core*).

Cada núcleo de uma CPU funciona como um processador independente, executando operações de apenas uma determinada tarefa por vez. Com a chegada dos processadores *multi core* houveram ganhos enormes no poder de processamento das máquinas, visto que seria possível executar mais de uma tarefa por vez ou até mesmo realizar uma mesma tarefa e dividi-la entre os diversos núcleos.

Entretanto, é necessário entender que mesmo contendo uma CPU *single core,* um dispositivo não tem todos seus recursos e outras operações travadas durante o processamento de uma determinada tarefa. Segundo Alecrim (2008), o que acontece é que "o processador dedica determinados intervalos de tempo a cada processo e isso acontece de maneira tão rápida, que se tem a impressão de processamento simultâneo". Por isso, na visão do usuário, todos os processos e aplicações parecem estar sendo executados ao mesmo tempo, mesmo em dispositivos *single core*.

Apesar de processadores *single core* criarem a falsa ilusão de execução simultânea, devido a velocidade de processamento, com o tempo os processos tornaram-se mais complexos e numerosos. Essa demanda trouxe ao mercado os processadores com múltiplos núcleos e, pode-se dizer que nos dias atuais é praticamente impossível encontrar um processador contendo apenas um núcleo.

Alecrim (2008) descreve um pouco do funcionamento de uma CPU *multi core* da seguinte maneira:

> CPUs deste tipo contam com dois ou mais núcleos distintos no mesmo circuito integrado, como se houvesse dois (ou mais) processadores dentro de um *chip*. Assim, o dispositivo pode lidar com dois processos por vez (ou mais), um para cada núcleo, melhorando o desempenho do computador como um todo.

Os processadores com múltiplos núcleos podem realizar mais tarefas de forma simultânea, e apesar de estarem compartilhando os recursos de um mesmo *chip*, podem operacionalizar de maneira individual, trabalhando em velocidades distintas por exemplo. Um núcleo pode funcionar em sua capacidade máxima, enquanto o outro trabalha em velocidade reduzida, gerando menos calor e até mesmo reduzindo o consumo de energia em alguns casos.

2.2. PROGRAMAÇÃO CONCORRENTE

Quando o assunto é a programação paralela (paralelismo), pode haver um ligeira confusão entre sua definição e a de programação concorrente (concorrência). Silva (1999) diz

que sistemas concorrentes devem lidar com atividades separadas que estão progredindo ao mesmo tempo. De maneira informal, é possível dizer que duas atividades são concorrentes se em algum momento elas encontram-se em algum ponto intermediário da execução, ou seja, ambas foram iniciadas, mas não terminadas. A autora ainda explica que a programação concorrente trabalha com atividades que estejam relacionadas de alguma forma, embora sejam executadas separadamente.

> Em alguns casos, as atividades estão relacionadas por terem um objetivo comum, por estarem cooperando para solucionar um problema. Em outros casos, a relação entre as atividades é mais frouxa, estas podendo ser totalmente independentes em seus propósitos, mas precisando compartilhar os recursos pelos quais competem por estarem executando em um ambiente comum.

Segundo a definição de Júnior (2011), a concorrência ocorre quando um servidor atende a vários dispositivos clientes escalando um determinado tempo para atender cada um, já o paralelismo ocorre quando vários servidores atendem vários clientes ao mesmo tempo, reduzindo o tempo de resposta para cada um deles.

2.3. PROCESSOS E THREADS

Sabe-se que a todo instante, computadores e dispositivos móveis estão executando programas e aplicativos diversos, sejam aqueles iniciados pelo usuário, ou outros que estão ativos em segundo plano realizando suas determinadas funções. Porém, as máquinas não executam somente esses programas que estão visíveis ao utilizador do dispositivo. Cada *software* pode possuir diversos processos relacionados a ele.

Amoroso (2009) faz uma explicação sucinta sobre o que seriam os processos e quais seriam suas funções dentro de um sistema.

> Simplificando, os processos representam tarefas em execução, mas nem todas têm relação direta com algum aplicativo. Muitas delas são executadas em pano de fundo e mantêm o sistema trabalhando - gerenciando redes, memória, disco, checagem antivírus, etc. Logo, podemos definir processos como softwares que executam alguma ação e que podem ser controlados de alguma maneira, seja pelo usuário, pelo aplicativo correspondente ou pelo sistema operacional.

Além dos processos, as *threads* realizam um papel importante dentro de um sistema computacional. As linhas (em inglês: *threads*) fazem com que um processo se autodivida em duas ou mais tarefas que podem ser executadas simultaneamente.

De acordo com o portal CanalTech (2014), uma *thread* pode ser definida como um subsistema. Seu uso permite que os processos se dividam em múltiplas tarefas que serão executadas em conjunto, resultando em um procedimento mais rápido em relação a um programa executado em um único bloco.

2.4. PROGRAMAÇÃO PARALELA

O paralelismo surgiu devido a evolução na arquitetura dos processadores. Como citado anteriormente, as CPU's atuais disponibilizam diversos núcleos que possibilitam que programas e aplicativos mais robustos possam ser executados com mais agilidade e eficiência. "O paralelismo em potencial consiste em um programa ou instrução que pode ser dividido em várias tarefas e cada uma destas pode ser processada em qualquer ordem, sem que isso altere o resultado final do programa" (CORREA, 2013). Uma outra definição, mais curta e objetiva, é feita por Santos (2015), a autora diz que um programa paralelo "é aquele que é simplesmente executado em vários processadores com o objetivo de melhorar a performance em relação à versão sequencial".

Em termos mais simples, a programação paralela utiliza vários recursos, neste caso, múltiplos processadores. Dessa forma, quando um problema é coletado, o mesmo é dividido em uma série de partes menores, fornecendo instruções para os diversos processadores (ou diversos núcleos) executarem as soluções ao mesmo tempo.

A programação paralela é uma abordagem que merece uma atenção dos desenvolvedores de *software,* visto que, além de um melhor aproveitamento do poder de processamento da máquina, outros benefícios podem ser destacados. O esforço reduzido de cada núcleo do processador e consequentemente um menor aquecimento do chip são exemplos positivos proporcionados pela utilização do paralelismo, visto que essa redução da carga de trabalho aumenta a vida útil da CPU e demais equipamentos de *hardware*, diminuindo a taxa de substituição ou até descarte desses componentes, colaborando diretamente com a preservação do meio ambiente.

Contudo, apesar das vantagens citadas anteriormente, colocar em prática a programação paralela é uma tarefa um pouco árdua, necessitando um esforço maior por parte do programador.

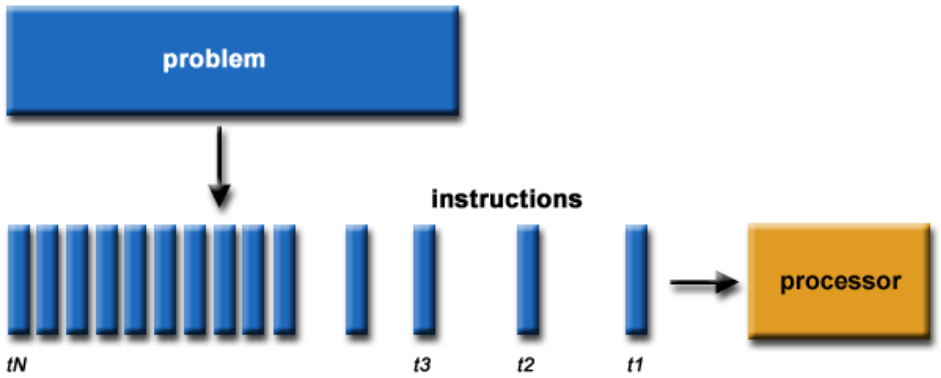
A programação paralela exige mais esforço do programador que a programação sequencial. “Escrever programas que utilizam paralelismo é um pouco diferente de escrever de maneira sequencial, como estamos mais acostumados a fazer” (JUNIOR, 2008). De acordo com o autor, devem ser definidas quais partes do programa serão executadas em paralelo e, além disso, é importante averiguar se há necessidade de estabelecer uma ordem de execução para as operações.

Em seu artigo que descreve sobre a utilização de programação funcional em paralelismo, Santos (2015) também destaca a maior dificuldade na utilização da programação paralela.

> Programação paralela é inerentemente mais difícil que programação sequencial. Tradicionalmente o programador não deve se preocupar apenas em descrever um algoritmo correto, mas também em como organizar as subtarefas na arquitetura em que a aplicação será executada.

Tradicionalmente, os softwares são escritos em sua grande maioria de forma sequencial, ou também, pode-se dizer que são escritos de forma serial. O problema a ser trabalhado é dividido em uma série de instruções, essas instruções são executadas uma após a outra pelo mesmo núcleo do processador. A Figura 1 demonstra de forma intuitiva como um problema (*problem*) é fragmentado em partes menores e enviado ao processador (*processor*).

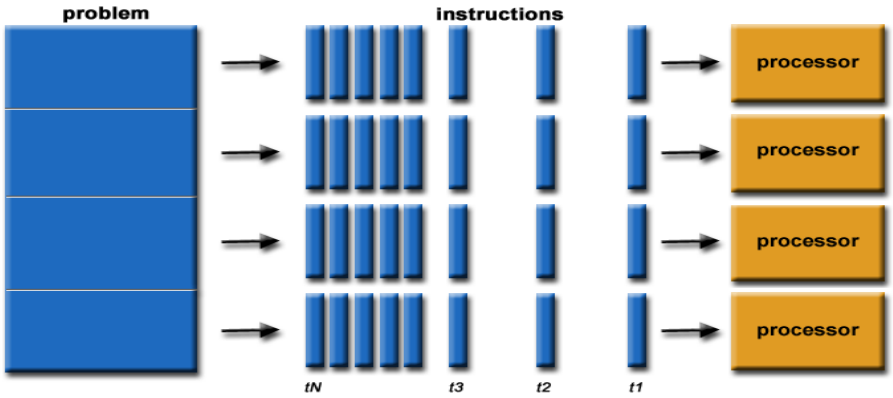
Figura 1 – Programação Sequencial



Fonte: Barney, 2018.

Na programação paralela, como citado anteriormente, os vários núcleos ou processadores disponibilizados pelos dispositivos são utilizados na resolução do problema. Dessa forma, o problema selecionado é dividido em partes que podem ser resolvidas simultaneamente e, em seguida, cada parte é subdividida em uma série de instruções que são executadas ao mesmo tempo pelos diferentes processadores. A Figura 2 ilustra de maneira simplificada o funcionamento do paralelismo.

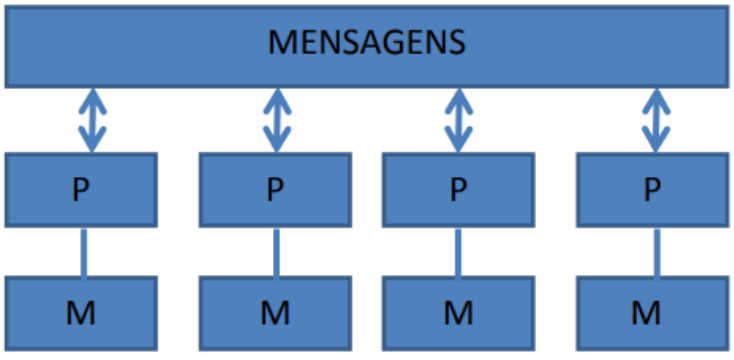
Figura 2 - Programação Paralela



Fonte: Barney, 2018.

Os s*oftwares* escritos com a utilização de programação paralela podem ser divididos em duas abordagens, sendo elas, o paralelismo por troca de mensagens e paralelismo por espaço compartilhado de memória. De acordo com Correa (2013), o método por troca de mensagens acontece quando processos estão alocados em processadores distintos e sem acesso a mesma área compartilhada de memória, dessa forma é necessário que haja uma troca de mensagens entre esses processos. O problema desse modelo é o tempo gasto para realizar a comunicação entre processos e memória, que pode ser resolvido com o uso de *buffer*, dessa forma evita-se essa custosa comunicação. A Figura 3 detalha o funcionamento do paralelismo por troca de mensagens, na qual P representa cada processo, enquanto M a memória, módulo de memória ou mesmo um dado ao qual esse processo esteja diretamente ligado.

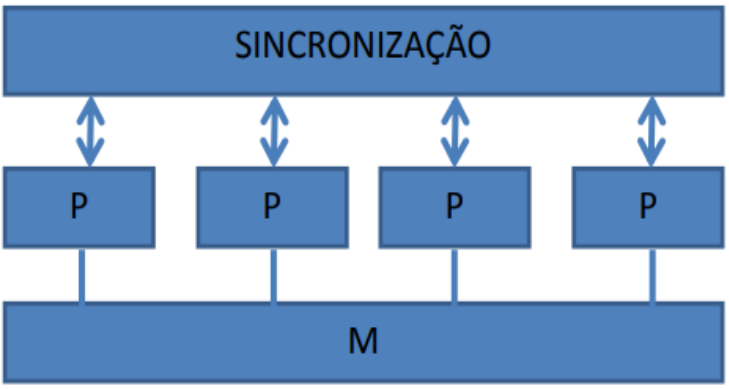
Figura 3 - Paralelismo por troca de mensagens



Fonte: Correa, 2013.

No paralelismo por espaço de memória compartilhada todos os processadores terão acesso a mesma área de memória. A Figura 4 demonstra o esquema dessa abordagem. Nessa ilustração P representa cada processo e M os dados, ou memória compartilhada, onde todos os processos P tem acesso.

Figura 4 - Paralelismo por espaço compartilhado de memória



Fonte: Correa, 2013.

Na abordagem demonstrada anteriormente, o problema pode ocorrer devido aos acessos a mesma memória. Correa (2013) diz que apesar da vantagem de se ter uma comunicação mais fácil e ágil, por outro lado, um ponto negativo são os possíveis comprometimentos aos resultados obtidos, visto que os processos irão alterar a mesma área de memória. Entretanto, assim como a abordagem por trocas de mensagens, a desvantagem do paralelismo por memória compartilhada também pode ser resolvida. Para solucionar o problema da segunda abordagem é necessário realizar uma sincronização dos processos e criar barreiras para que dois ou mais processos não alterem a mesma área de memória ao mesmo tempo.

2.5. LINGUAGEM JAVASCRIPT

Pode-se dizer que é incomum estar inserido no “mundo da programação” e nunca ter sequer visto algo ou alguém mencionar a linguagem de programação *JavaScript*. Seja uma pessoa que trabalha em outro nicho da programação ou mesmo um estudante iniciante em um curso de computação, a maioria deles provavelmente já ouviram falar de JS em algum momento.

Em sua publicação no blog de tutoriais da Hostinger, Zampieri (2019) diz que o *JavaScript* é uma linguagem de programação bastante utilizada e extremamente importante no desenvolvimento de *websites* e sistemas que funcionam na *web*. De acordo com o ranking da TIOBE (2021), a linguagem ocupa a sétima posição entre as linguagens de programação mais populares no mundo.

Nascida na década de 90, mais precisamente em 1996, a linguagem JS foi criada por Brendan Eich para funcionar no navegador *Netscape Navigator* da *Netscape Communications Corporation*, na qual ele era funcionário. O objetivo de Brendan era facilitar processos dentro de páginas web, descomplicando a programação de animações e alertas, tornando o processo mais fácil e ágil.

Aproximadamente um ano pós criação, o JS começou a ser inserido no navegador da gigante *Microsoft*, o que alavancou o nome da linguagem e sua força no mercado. Daquele tempo em diante o crescimento não parou e, nos dias atuais, a linguagem pode ser considerada como uma das mais robustas e versáteis, possuindo diversas funcionalidades e sendo utilizada também para a criação de aplicativos em dispositivos móveis.

2.6. PARALELISMO EM JAVASCRIPT

Mesmo sendo uma linguagem que proporciona ao desenvolvedor diversas possibilidades, o JS apresenta certas barreiras quando o objetivo é a criação de uma aplicação que necessite executar *scripts* em paralelo. Esse fator negativo é proveniente da própria linguagem, isso ocorre devido ao fato de o JS ser uma linguagem de encadeamento único.

Isso significa que, quando um *script* é colocado em execução, todo o restante da página *web* deixa de responder, e mesmo que na maioria dos casos esses congelamentos não sejam perceptíveis ao usuário, devido ao poder de processamento dos navegadores modernos, a existência desse problema não pode ser ignorada. Quando a execução do *script* exige mais recursos e mais tempo de processamento, esses gargalos podem afetar de forma significativa o desempenho da aplicação e a experiência do usuário.

Hiwarale (2018) descreve a respeito dos problemas apresentados pelo encadeamento único, ou seja, o fato de linguagens de *thread* único executarem apenas uma instrução por vez, usando o *JavaScript* como instrumento de sua pesquisa.

> Sempre que você faz algo em *JavaScript* (por exemplo, inserir texto dentro de um elemento DOM), toda a página da web congela e deixa de responder. Isso significa que quando seu navegador está executando um script, todas as outras operações (como manipulação de DOM, animação, desenho, execuções adiadas e outras operações que acontecem no thread principal) serão interrompidas.

Existe uma saída para superar essa barreira característica da linguagem, trata-se de uma API (*Application Programming Interface*) introduzida pelo HTML5 denominada *Web Workers*.

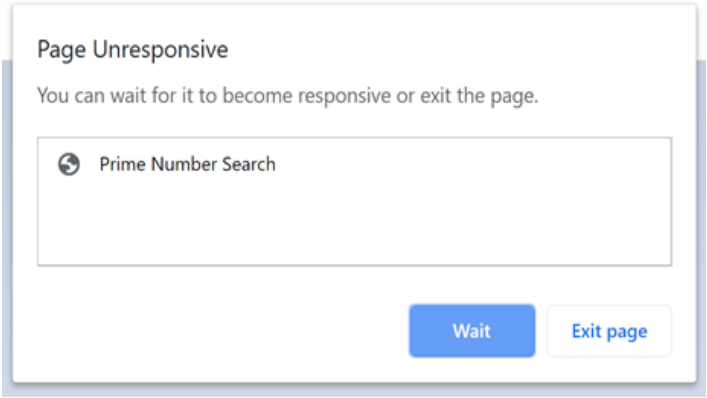
Esta solução surgiu devido a um esforço colaborativo entre Mozilla e Google para fortalecer ainda mais seus navegadores.

Um *web worker* pode ser definido como um *JavaScript* que executa em segundo plano, sem afetar o desempenho da página, ou ainda, como um script rodando em uma *thread* diferente, simultaneamente com a *thread* principal. O funcionamento dos *web workers* é descrito por Hiwarale (2018) da seguinte maneira:

> O navegador cria uma *thread* por guia. A *thread* principal pode gerar um número ilimitado de *web workers*, até que os recursos do sistema do usuário sejam totalmente consumidos. Assim que uma guia do navegador é fechada, a *thread* principal 'morre' e 'mata' todos os *web workers* que ela gerou. Uma *thread* principal pode 'matar' *web workers* que ele gerou, e um *web worker* pode 'matar' a si mesmo. Quando um *web worker* morre, os fios anexados a ele morrem também.

Um cenário ideal para a utilização dos *web workers* seria uma página que apresenta aquela famosa mensagem de "Página sem resposta" durante sua execução, isso acontece devido ao fato de o *script* ser um pouco mais (ou muito mais) complexo do que se imagina. A Figura 5 mostra o problema citado anteriormente, no qual a página não responde por estar esperando o fim da execução do *script*.

Figura 5 - Página sem resposta



Fonte: MacDonald, 2019.

Basicamente, o uso de programação paralela em *JavaScript*, ou em qualquer outra linguagem, torna-se justificável quando tem-se um problema mais complexo ou de cálculos com maiores proporções. Caso contrário, torna-se dispensável, visto que sua implementação é mais complicada e navegadores modernos executam tão rapidamente os *scripts* mais modestos, que os usuários acabam não percebendo nenhum congelamento em suas páginas.

2.7. FERRAMENTAS E TECNOLOGIAS

Para o desenvolvimento do presente trabalho fez-se necessário o uso de algumas ferramentas e tecnologias. A seguir estão listadas essas ferramentas e tecnologias que foram de extrema importância no desenvolvimento deste estudo.

2.7.1. HTML

Criada por Tim Bernes-Lee em 1991, a Linguagem de Marcação de Hipertextos, do inglês *HyperText Markup Language* (HTML) é o componente básico da web, ele estabelece a estrutura básica de um *website*, além da inserção de conteúdo. Os navegadores, como Chrome, Firefox, Safari e diversos outros são os responsáveis por fazer a interpretação desses arquivos HTML.

Segundo Marques (2019), essa linguagem de marcação serve para dar significado e organizar as informações de uma página na web. Sem o HTML, o navegador não saberia exibir textos como elementos ou carregar imagens e outros conteúdos. Em uma analogia ao corpo humano, o HTML poderia ser considerado o esqueleto.

2.7.2. CSS

No ano de 1995, Hakon Wium Lie e Bert Bos apresentaram a proposta do *Cascading Style Sheets* (CSS) que logo foi apoiada pela *World Wide Web Consortium* (W3C). O intuito aqui era dar uma aparência melhor para a página HTML, que por si só, é composta de elementos com pouca qualidade visual. Marques (2019) diz que com a utilização do CSS podemos estilizar todos os elementos, aplicando espaçamentos, cores, posicionamentos, tamanho de fontes, famílias de fontes, bordas e outros efeitos visuais que dão forma ao documento. O CSS seria aquilo que cobre o esqueleto (pele, cabelos, coloração, etc.) e dá características a ele, se pensarmos na analogia anteriormente citada.

2.7.3. Javascript

O *Javascript* é o responsável por disparar ações e dar vida as páginas *web*, foi criado em 1996 por Brendan Eich e diferentemente do HTML e CSS é considerado uma linguagem de programação. De acordo com Marques (2019), o JS é o responsável por tornar os elementos mais dinâmicos, ele é quem permite a execução de *scripts* na página *web*. Dessa forma, quando o usuário está navegando em uma página e executa uma ação que resulte em um comportamento na página, muito provavelmente, esse comportamento foi executado pelo JS.

2.7.4. Visual Studio Code

O *Visual Studio Code*, ou simplesmente VSCode, foi lançado em 2015 pela *Microsoft*. O intuito da empresa foi criar um editor de código leve e multiplaforma destinado ao desenvolvimento de aplicações *web*, porém, nos dias atuais o editor abrange uma gama bem maior de possibilidades e linguagens.

Atualmente, é possível até mesmo realizar a programação de aplicações em Python, Ruby e C++. A ferramenta conta com uma enorme variedade de extensões, quando instaladas e ativadas, essas extensões auxiliam de maneira significativa o trabalho do programador.

2.7.5. Navegadores baseados no projeto Chromium

Um *browser* ou navegador (em português) é um *software* criado para permitir a navegação pela internet, servindo como uma ponte de acesso aos *websites* disponibilizados pela rede mundial de computadores. Neste estudo, foram utilizados três navegadores, sendo eles: Chrome, Opera e Edge. Essa escolha ocorreu devido ao fato de serem navegadores modernos, criados por empresas renomadas e por serem todos baseados no projeto Chromium. De acordo com o Chromium *Projects* (2008), o projeto Chromium consiste em um navegador de código aberto que visa criar uma maneira mais segura, rápida e estável de todos os usuários da internet experimentarem a *web*.

2.7.6. XAMPP

O XAMPP é um *software* gratuito e multiplataforma que inclui os principais servidores de código aberto. Através dele, é possível simular localmente um servidor *web* de maneira simples e dinâmica. Uma solução muita prática a disposição dos desenvolvedores.

De acordo com o Apache Friends (2016), o objetivo do XAMPP é construir uma distribuição de fácil instalação, configurada com todos os recursos necessários já ativados. Dessa forma, desenvolvedores podem ingressar no "mundo do Apache" sem muitas dificuldades.

2.8. TRABALHOS CORRELATOS

Pode-se afirmar que o estudo da programação paralela ainda carece de muita pesquisa e disponibilidade de informações, principalmente quando a linguagem utilizada é o JS. Apesar de estar pouco presente em nosso meio acadêmico, foi possível encontrar alguns trabalhos com características similares e extremamente importantes para o desenvolvimento dessa pesquisa.

Disponibilizado no repositório online de tutoriais do Laboratório Nacional Lawrence Livermore, o trabalho de Barney (2018), intitulado "*Introduction to Parallel Computing Tutorial*" foi primordial na construção desta pesquisa. O material contém diversas informações sobre o paralelismo, desde seu funcionamento na perspectiva do *software* quanto também no segmento de *hardware* e, além disso, apresenta ao leitor os diversos cenários e situações que fazem com que o paralelismo seja praticamente indispensável para obtenção de resultados satisfatórios. Mesmo disponibilizando um documento com tantas informações, o autor afirma que o intuito de seu estudo é fornecer apenas uma visão geral e introdutória acerca do paralelismo, o que basicamente também é a ideia deste trabalho.

Entretanto, o trabalho de Barney carece de elementos e codificações que demonstrem na prática como a utilização do paralelismo pode ser relevante no desenvolvimento de *softwares* e na otimização dos processos realizados por eles. Esta pesquisa, diferentemente do trabalho de Barney, buscou mostrar na prática o funcionamento de uma aplicação paralelizada, apresentando dados comparativos em relação à uma aplicação construída sequencialmente. Esses fatores são importantes principalmente para aqueles que estão estabelecendo seu primeiro

contato com a abordagem, afinal, visualizar resultados é extremamente importante para gerar interesse e engajar o leitor.

A solução computacional desenvolvida neste trabalho teve como maior desafio a utilização da programação paralela com JS, afinal, trata-se de uma linguagem de *thread* único. A publicação "*A Simple Introduction to Web Workers in JavaScript*" de Matthew MacDonald mostra como é possível paralelizar JS através da utilização da API *web workers*. Além da fundamentação do assunto abordado, o autor implementa um código em dois cenários, o primeiro sem usar os *web workers* e em seguida utilizando a API.

Em seu experimento, ao executar um código que busca por números primos em um intervalo grande, a página congela e deixa de responder por alguns instantes. Ao utilizar os *web workers* o problema é solucionado. MacDonald (2019) afirma que para visualizar os benefícios dos *web workers* é preciso executar um código que demande mais tempo de execução, caso contrário, a diferença não é perceptível.

No estudo desenvolvido por MacDonald, a utilização de uma solução que trabalha com matemática simples faz com que o leitor possa entender a codificação sem muitos problemas, visto que a lógica por trás dos números primos é trivial para grande parte das pessoas. No entanto, pode-se dizer que um exemplo tão simples pode não explicitar a dimensão daquilo que pode ser feito ou aperfeiçoado com a utilização do paralelismo.

Buscando trazer uma visão um pouco mais próxima dos problemas reais da computação, o estudo de caso utilizado nesta pesquisa trouxe algo que geralmente pode tornar-se bastante complexo para os desenvolvedores, exigindo mais esforços durante a codificação e consumindo recursos consideráveis das máquinas. Apesar de simples, o caso presente nesse estudo baseia-se no processamento e tratamento de imagens, um problema que pode possuir um grau de complexidade alto dependendo da demanda.

De fato, são poucos os estudos disponibilizados, principalmente se tratando da paralelização de códigos em linguagem *JavaScript.* No entanto, os trabalhos de Barney e McDonald devem ser exaltados por contribuírem de forma tão significativa para o estudo da programação paralela. A forma didática ao apresentar o assunto abordado torna-se extremamente importante para quem está realizando o primeiro contato com a temática.

3. METODOLOGIA

Este capítulo aborda a natureza da pesquisa, bem como a identificação do seu caráter, métodos e procedimentos e o tratamento dos dados.

3.1. NATUREZA DA PESQUISA

A metodologia aplicada no desenvolvimento deste estudo contém caráter qualitativo. De acordo com Dalfovo (2008), o método qualitativo é trabalhado predominantemente com informações coletadas pelo pesquisador, sendo esta não expressa em números, ou quando existentes, esses números e as conclusões neles baseadas representam menor interferência na análise.

No entanto, o caráter dessa pesquisa é quantitativo, pois os dados obtidos serão dimensionados e comparados. De acordo com Diana (2017), a pesquisa quantitativa é usada para quantificar um problema por meio da geração de dados numéricos ou dados que podem ser transformados em estatísticas utilizáveis.

3.2. MÉTODOS E PROCEDIMENTOS

O objetivo inicial do presente trabalho foi disseminar a ideia acerca da utilização da programação paralela e sua importância no desenvolvimento de *softwares* que operam com problemas mais complexos. Este trabalho visou atingir, principalmente, aqueles que estão tendo seu primeiro contato com a abordagem.

Para reforçar os argumentos apresentados, foi construída uma aplicação utilizando programação sequencial e, posteriormente, outras versões dessa mesma aplicação utilizando paralelismo. O processo de desenvolvimento dessas soluções seguiu algumas etapas, sendo elas: a seleção do problema a ser trabalhado, a realização de um estudo mais amplo a respeito dos *web workers* e, por fim, a coleta e apresentação dos dados provenientes da realização de testes de desempenho.

A primeira etapa consistiu na seleção de um estudo de caso que sustentasse a proposta do trabalho. O objetivo nessa etapa foi a obtenção de uma demanda que poderia ser melhorada com a utilização do paralelismo. Através de pesquisas e discussões com o orientador, um problema com processamento e tratamento de imagens foi selecionado e utilizado no desenvolvimento de uma solução computacional.

Após a seleção do estudo de caso, foi iniciada a etapa de desenvolvimento das aplicações, que aconteceu em consonância com um estudo mais profundo a respeito dos *web workers*, haja vista que se tratava de um assunto pouco "familiar" aos alunos elaboradores do projeto. Nessa etapa, foram utilizadas algumas das ferramentas e tecnologias apresentadas anteriormente, sendo elas: HTML, CSS, JS, VSCode e XAMPP.

Por fim, a última etapa consistiu na realização de testes nas aplicações e coleta dos resultados obtidos. As soluções foram colocadas em execução e comparadas através das ferramentas de performance disponibilizadas pelos navegadores Chrome, Opera e Edge, escolhidos por fatores já descritos neste trabalho.

Dessa forma, foram obtidas informações como: tempos de execução de *scripts*, dados de consumo de CPU e duração de bloqueios ou congelamentos. Esses critérios foram selecionados devido ao fato de apresentarem diferenças visíveis em relação às duas abordagens comparadas, visto que, trata-se de uma solução simples e, alguns outros parâmetros poderiam não explicitar tão claramente essas diferenças de desempenho.

3.3. TRATAMENTO DOS DADOS

Após a finalização das etapas de desenvolvimento e teste das aplicações, houve um tratamento dos dados. Estas informações foram obtidas, principalmente, através da realização de uma média aritmética entre os números coletados. Esses dados foram provenientes de medições de desempenho e consumo de recursos computacionais, realizados nas ferramentas de performance dos navegadores baseados no projeto Chromium. Os dados coletados servem como apoio à conclusão do trabalho que, buscou mostrar se houveram vantagens no uso da programação paralela na aplicação desenvolvida com linguagem *JavaScript*.

4. RESULTADOS E DISCUSSÃO

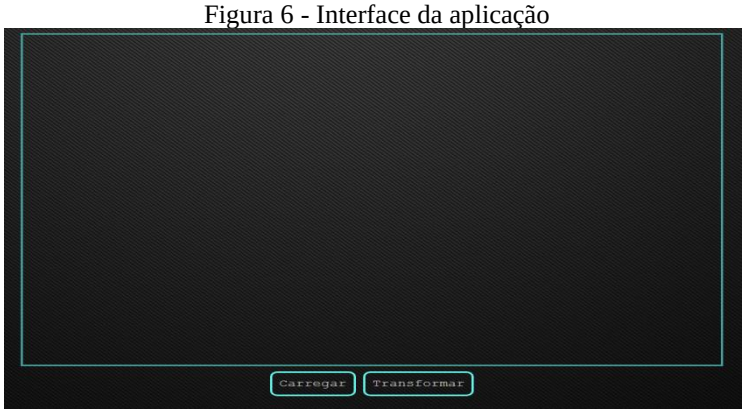
Esse capítulo apresenta um resumo sobre o funcionamento das soluções desenvolvidas em *JavaScript* e os resultados obtidos através de testes de desempenho e suas devidas comparações.

4.1. DESCRIÇÃO DA APLICAÇÃO

Apesar de ser um problema que muitas vezes pode ser extremamente complexo e necessitar de um esforço maior por parte dos programadores, o processamento e tratamento de imagens foi abordado de forma simplificada nesta pesquisa. Por se tratar de um trabalho exploratório, o intuito foi desenvolver algo de fácil compreensão e que conseguisse apoiar a tese apresentada, mostrando se haveriam ganhos de desempenho através da paralelização do *software* desenvolvido.

A função da solução construída é realizar a aplicação de um filtro de cor cinza nas imagens inseridas pelo usuário. Foram desenvolvidas algumas versões dessa mesma aplicação, possuindo interfaces e funcionalidades idênticas, porém, com distinções em suas codificações e sendo distribuídas em cinco versões. A Figura 6 apresenta a interface da solução.

Figura 6 - Interface da aplicação



Fonte: Elaborada pelos autores.

A primeira dessas versões foi construída sem a paralelização do *script* e todo o processo de aplicação do filtro sobre a imagem acontece na *thread* principal. Já as outras quatro versões tiveram seus *scripts* paralelizados com a utilização dos *web workers*, contendo respectivamente

1, 2, 4 e 8 *workers*, fazendo com que a carga de trabalho exercida pela aplicação do filtro seja dividida entre eles.

Ao iniciar a aplicação, o usuário deve clicar no botão “Carregar”, selecionar uma imagem de seu computador e esperar que a mesma seja carregada na tela. Em seguida, o botão “Transformar” precisa ser acionado para que o *software* aplique o filtro de cor cinza sobre a imagem. Após isso, o processo está finalizado.

Vale ressaltar que a ideia do problema descrito foi fornecer informações capazes de mostrar como a paralelização de um código altera seu tempo de execução e a quantidade de recursos computacionais consumidos por ele. Dessa forma, pode-se dizer que essa pequena solução se enquadrou como uma espécie de “cobaia” para realização dos testes presentes neste estudo. As Figuras 7 e 8 demonstram o antes e depois da aplicação do filtro sobre a imagem.

Figura 7 - Imagem antes do filtro

Fonte: Elaborada pelos autores.

Figura 8 - Imagem depois do filtro

Fonte: Elaborada pelos autores.

4.2. CODIFICAÇÃO

Esta seção tem como intuito descrever resumidamente as codificações e a lógica de programação utilizada na construção das aplicações desenvolvidas. Todas as versões dessas soluções são compostas por arquivos de estilização idênticos, as distinções ficam por parte do código HTML (alterado de acordo com a necessidade do *software*) e pela presença do arquivo "worker.js" nas versões paralelizadas.

No início da seção "<*body*>" são estabelecidas as codificações para realização do carregamento da imagem que receberá o filtro de cor cinza. A tag "<canvas>" é utilizada para desenhar a imagem selecionada. De acordo com a W3Schools (2012), o canvas é um elemento HTML usado para desenhar gráficos em tempo real através de linguagem *JavaScript*. A Figura 9 demonstra o trecho citado.

Figura 9 - Trecho de código - Parte 1

```
<body>
    <div class="bg">
        <div id="titulo">
            <h1>TCC-II</h1>
        </div>
        <div id="imagem">
            <canvas id="preview"></canvas>
        </div>
        <div id="conteudo">
            <input type="file" accept="image/*" name="input" id="input">
            <label for="input">Carregar</label>
            <button id="button">Transformar</button>
        </div>
    </div>
```

Fonte: Elaborada pelos autores.

É possível dizer que, em um primeiro momento, o <*canvas*> se parece um pouco com o elemento <*img*>, responsável pela inserção de imagens em documentos HTML. No entanto, o que acontece é que o canvas reserva um espaço, uma espécie de superfície para desenho (inicialmente toda em branco) esperando algo desenhar sobre ela.

Após a definição dessa área de desenho, os *scripts* serão os responsáveis por desenhar nela. O primeiro passo é a obtenção dos elementos necessários através do método "*document.getElementById()*". Posteriormente, deve ser definido pelo método "*HTMLCanvasElement.getContext()*", qual o contexto do desenho a ser construído, podendo ser em 2D ou 3D, por exemplo. A Figura 10 apresenta o trecho citado.

Figura 10 - Trecho de código - Parte 2

```
<script type="text/javascript">
    const $input = document.getElementById('input')
    const $preview = document.getElementById('preview');
    const previewCtx = $preview.getContext('2d');
    const image = new Image();
```

Fonte: Elaborada pelos autores.

De acordo com o MDN Mozilla (2020), o "*HTMLCanvasElement.getContext( )*" é utilizado para obter o contexto de renderização e suas funções de desenho, recebendo como parâmetro o tipo do contexto e retornando *null* caso o mesmo não seja suportado ou tenha sido configurado de maneira diferente. No estudo de caso apresentado nesta pesquisa, foi definida a utilização de imagens dentro de um contexto 2D.

Após a definição do contexto, o *script* começa a realização de suas tarefas dentro do espaço definido pelo canvas. No problema utilizado neste trabalho, esse *script* desenha a imagem selecionada pelo usuário e faz a aplicação de um filtro de cor cinza sobre ela. Essas operações ocorrem através da manipulação da matriz que representa a imagem, na qual, laços de repetição percorrem pixel a pixel todo o desenho, fazendo com que as diversas cores do padrão RGB sejam substituídas por um tom de cinza, resultando em uma imagem totalmente alterada. Essa modificação é obtida a partir da média aritmética das três cores do padrão RGB. O trecho descrito é demonstrado a seguir na Figura 11.

Figura 11 - Trecho de código - Parte 3

```
function applyFilter() {
    console.time()
    const imageData = previewCtx.getImageData(0, 0, $preview.width, $preview.height)
    for (let x = 0; x < imageData.width; x++) {
        for (let y = 0; y < imageData.height; y++) {
            let index = (x + (y * imageData.width)) * 4
            var red = imageData.data[index]
            var green = imageData.data[index + 1]
            var blue = imageData.data[index + 2]
            var gray = (red + green + blue) / 3;

            imageData.data[index] = gray;
            imageData.data[index + 1] = gray;
            imageData.data[index + 2] = gray;
        }
    }
    previewCtx.putImageData(imageData, 0, 0)
    console.timeEnd()
}
```

Fonte: Elaborada pelos autores.

A diferença entre o código não paralelizado e os demais (paralelizados com *web workers*) resume-se, basicamente, na maneira como ocorre a aplicação do filtro de cor cinza. Na versão sem paralelização, todo o código está contido dentro do arquivo HTML e o processo

de transformação da imagem ocorrerá na *thread* principal. Nas demais versões (paralelizadas), o código de aplicação do filtro está contido em um arquivo *JavaScript* que representa o *web worke*r, no qual as operações que alteram as cores da imagem estarão acontecendo em outras *threads*, paralelamente à principal.

Um *worker* é criado através do construtor "*Worker()*", que carrega um arquivo *Javascript* para execução do seu código em uma *thread*. Segundo o MDN Mozilla (2019), a *thread* principal e as *threads* secundárias (representadas pelos *workers)* comunicam-se entre si enviando dados através de um sistema de mensagens. Ambos os lados enviam essas mensagens usando o método "*Worker.postMessage()*", as respostas são obtidas via um manipulador de eventos. É importante ressaltar que os dados são copiados, e não compartilhados.

O intuito deste capítulo foi demonstrar, sucintamente, a lógica utilizada na programação das soluções. Todos os códigos e arquivos utilizados poderão ser acessados e baixados por aqueles que tenham interesse em realizar algum estudo ou procedimento similar à esta pesquisa, os arquivos estão disponíveis para download no link https://github.com/heronlima/Paralelismo-JS-WebWorkers.

4.3. TESTES E COMPARAÇÕES

Esta seção descreve a etapa posterior à produção das soluções computacionais, na qual ocorreram os testes de desempenho e comparações entre as versões da aplicação, tendo como objetivo averiguar se a paralelização dos códigos desenvolvidos se mostrou uma alternativa viável.

Para consumir uma quantidade considerável de recursos, foi utilizada uma imagem no formato JPG de alta definição e resolução, possuindo proporções de 10000x7000 pixels e aproximadamente 8,5Mb de tamanho de armazenamento. Como plataforma para testes, foi utilizado um computador *desktop* equipado com um processador AMD Ryzen 7 2700 contendo 8 núcleos e 16 *threads*, 16 Gb de memória RAM DDR4, placa de vídeo NVIDIA Geforce GTX 1660 e sistema operacional Windows 10 Pro de 64 bits. Vale salientar que, não é necessária uma máquina com tais configurações para a realização destes testes. Um computador que atenda aos requisitos recomendados descritos pelos navegadores modernos, provavelmente, será suficiente para execução e realização dos testes demonstrados.

Para execução da solução, tornou-se necessário simular localmente um servidor web e, para tal, foi utilizado o *software* gratuito XAMPP. Na etapa de testes de desempenho foram utilizadas as ferramentas do desenvolvedor disponibilizadas dentro dos próprios navegadores Chrome, Edge e Opera. Esses navegadores foram escolhidos por serem baseados em um mesmo projeto, logo, suas ferramentas de performance são praticamente iguais e apresentam os mesmos tipos de dados, facilitando a coleta dos mesmos.

Todos os testes foram realizados no modo anônimo dos navegadores, dessa forma, as extensões e *plugins* dos mesmos produzem uma menor interferência no desempenho de cada *browser*. Os parâmetros avaliados nessa primeira parte dos testes foram:

a) Tempo total de execução de *scripts*;
b) Tempo total de bloqueio (TBT);
c) Tempo de execução do *script* de aplicação do filtro cinza.

As duas primeiras informações foram obtidas através das ferramentas de desempenho dos navegadores, já o terceiro parâmetro, foi coletado pela função "*console.time*()", implementada diretamente no código.

Segundo o MDN Mozilla (2019), o "*console.time()*" inicia um cronômetro que monitora o tempo que uma operação leva, apresentando após a utilização de "*console.timeEnd()*" o tempo, em milissegundos, que se passou desde que o cronômetro iniciou.

Já o TBT é definido por Walton (2020) como o tempo decorrido, no qual *Long Tasks* (todas as tarefas com mais de 50ms) bloquearam a *thread* principal e afetaram a usabilidade da página *web*.

Os dados descritos em cada parâmetro avaliado foram obtidos após a realização da média aritmética entre três execuções. A Tabela 1 demonstra as informações coletadas após os testes aplicados na solução que não possuía o código paralelizado.

Tabela 1 - Aplicação sem utilização de *web workers*

PARÂMETRO	CHROME Tempo decorrido (ms)	EDGE Tempo decorrido (ms)	OPERA Tempo decorrido (ms)
Tempo de *script* (*thread* principal)	6214,67	6055,33	6725,00
Tempo total de bloqueio	6173,25	6011,13	6682,07
Tempo de *scripts* (via *console.time*)	6206,18	6047,00	6716,15

Fonte: Elaborado pelos autores.

Após visualizar a tabela acima, é possível perceber que, mesmo tratando-se de uma aplicação simples, sendo executada em máquina com boas especificações de *hardware*, foi necessário um tempo razoável para que todo o processo chegasse ao fim. A duração do TBT e execução dos scripts fazem com que o usuário tenha que esperar a aplicação do filtro antes de interagir de alguma outra forma com a página.

Com o intuito de otimizar essa aplicação, reduzindo o tempo de suas operações, foram criadas mais quatro versões, todas paralelizadas através de *web workers*. As Tabelas 2 e 3 demonstram as informações coletadas após a realização de testes nessas versões.

Tabela 2 - Aplicação utilizando 1 *web worker*

PARÂMETRO	**CHROME Tempo decorrido (ms)**	**EDGE Tempo decorrido (ms)**	**OPERA Tempo decorrido (ms)**
Tempo de *script* (*thread* principal)	448,00	529,00	754,00
Tempo de *script* (1 *worker*)	1075,00	1074,00	1143,00
Tempo de *script* (*thread* principal + *workers*)	1523,00	1603,00	1897,00
Tempo total de bloqueio	361,39	424,49	665,52
Tempo de *scripts* (via *console.time*)	1414,11	1560,08	1827,01

Fonte: Elaborado pelos autores.

Tabela 3 - Aplicação utilizando 2 *web workers*

PARÂMETRO	**CHROME Tempo decorrido (ms)**	**EDGE Tempo decorrido (ms)**	**OPERA Tempo decorrido (ms)**
Tempo de *script* (*thread* principal)	494,00	724,00	942,00
Tempo de *script* (2 *workers*)	1211,00	1183,00	1268,00
Tempo de *scripts* (*thread* principal + *workers*)	1705,00	1907,00	2210,00
Tempo total de bloqueio	404,21	622,37	790,06
Tempo de *scripts* (via *console.time*)	936,22	1103,68	1512,97

Fonte: Elaborado pelos autores.

Após a visualização dos dados apresentados nas Tabelas 2 e 3, é possível perceber como os tempos de execução foram reduzidos. O somatório entre a duração dos processos realizados na *thread* principal (*Main Thread*) e *threads* secundárias (*workers*) é um valor bem menor em relação aos apresentados pela primeira versão da aplicação.

A paralelização dos códigos poderia ter sido encerrada após a obtenção das informações presentes nas Tabelas 2 e 3, visto que uma melhora de desempenho expressiva já havia sido obtida. Todavia, mais duas versões foram implementadas. Dessa forma, seria possível averiguar se uma quantidade maior de *workers* melhorariam ainda mais o desempenho da aplicação. As Tabelas 4 e 5 apresentam os dados obtidos após testar as versões contendo 4 e 8 *workers*, respectivamente.

Tabela 4 - Aplicação utilizando 4 *web workers*

PARÂMETRO	**CHROME Tempo decorrido (ms)**	**EDGE Tempo decorrido (ms)**	**OPERA Tempo decorrido (ms)**
Tempo de *script* (*thread* principal)	681,00	1031,00	1239,00
Tempo de *script* (4 *workers*)	1326,00	1260,00	1400,00
Tempo de *scripts* (*thread* principal + *workers*)	2007,00	2291,00	2639,00
Tempo total de bloqueio	492,14	808,16	1057,99
Tempo de *scripts* (via *console.time*)	976,11	1292,06	1552,69

Fonte: Elaborado pelos autores.

Tabela 5 - Aplicação utilizando 8 *web workers*

PARÂMETRO	**CHROME Tempo decorrido (ms)**	**EDGE Tempo decorrido (ms)**	**OPERA Tempo decorrido (ms)**
Tempo de *script* (*thread* principal)	681,00	1031,00	1239,00
Tempo de *script* (4 *workers*)	1326,00	1260,00	1400,00
Tempo de *scripts* (*thread* principal + *workers*)	2007,00	2291,00	2639,00
Tempo total de bloqueio	492,14	808,16	1057,99
Tempo de *scripts* (via *console.time*)	976,11	1292,06	1552,69

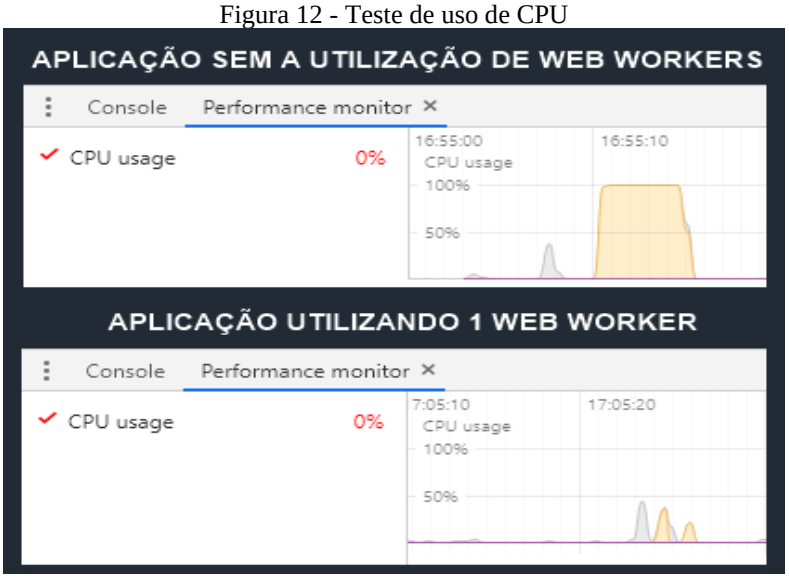
Fonte: Elaborado pelos autores.

Visualizando os dados apresentados anteriormente nas Tabelas 4 e 5, foi possível concluir que na aplicação desenvolvida, a implementação de quatro e oito *workers* mostrou-se um esforço desnecessário, visto que a duração das operações aumentou (em relação as versões contendo um ou dois *workers*) e os programadores tiveram que demandar um tempo maior para implementação das codificações.

Ainda em relação aos diagnósticos de desempenho, foi possível coletar informações relacionadas ao uso de CPU (CPU *Usage*). Tendo em vista a grande similaridade entre as ferramentas de desempenho dos três navegadores, este teste de CPU *usage* foi realizado apenas no Chrome.

Os códigos paralelizados demonstraram também resultados bem similares quando comparados entre si, dessa forma, os dados apresentados a seguir na Figura 11 fazem um paralelo apenas entre a versão não paralelizada e a versão contendo um *web worker*.

Figura 12 - Teste de uso de CPU



Fonte: Elaborado pelos autores.

É possível notar a grande melhoria de desempenho obtida pela aplicação após a paralelização de seu código. O consumo de CPU, que atingiu 100% de uso na primeira versão, foi reduzido à uma taxa abaixo dos 50% nas versões contendo *web workers*.

Tendo em vista os aspectos observados, foi feito um balanço geral do desempenho apresentado por cada uma das versões da aplicação. A Tabela 6 apresenta dados relacionados ao tempo utilizado na execução de suas respectivas operações.

Tabela 6 - Comparação entre todas as versões da aplicação

PARÂMETRO	**Sem *workers* Tempo decorrido (ms)**	**1 *worker* Tempo decorrido (ms)**	**2 *workers* Tempo decorrido (ms)**	**4 *workers* Tempo decorrido (ms)**	**8 *workers* Tempo decorrido (ms)**
Tempo total de *scripts (Thread* principal + *workers)*	6331,67	1674,33	1940,67	2312,33	3292,67
Tempo total de bloqueio	6288,82	483,80	605,55	786,10	954,89
Tempo de *scripts* (via *console.time*)	6323,11	1600,40	1184,29	1273,62	1455,99

Fonte: Elaborado pelos autores.

Visando estimar em termos percentuais o quão significativos se mostraram os ganhos obtidos pelas versões paralelizadas, foram levantados dados que explicitaram o quão defasada tornou-se a primeira versão da aplicação. A Tabela 7 demonstra os resultados coletados após a realização de um comparativo entre a versão (sem adição de *workers)* e a versão contendo um único *worker*.

Tabela 7 - Percentuais de melhoria de desempenho

PARÂMETRO	**Taxa de tempo reduzido após a utilização de 1 *worker***
Tempo total de *scripts (Thread* principal + *workers)*	278%
Tempo total de bloqueio	1200%
Tempo de *scripts* (via *console.time*)	295%

Fonte: Elaborado pelos autores.

Os dados demonstram o quão expressivas foram as reduções de tempo de execução obtidas a partir da paralelização do código. A execução dos *scripts* consumiu aproximadamente 3x (três vezes) menos tempo e TBT reduziu seu consumo em 12x (doze vezes).

5. CONSIDERAÇÕES FINAIS

O intuito deste trabalho foi produzir um estudo exploratório que abordasse principalmente a paralelização de códigos *JavaScript*. Acredita-se que a área de pesquisa foi enriquecida, visto que é uma abordagem ainda pouco estudada e com escassez de conteúdo, principalmente em língua portuguesa.

Apesar de apresentar um *software* simples como objeto de estudo, ressalta-se que o objetivo principal foi demonstrar se a paralelização dos códigos resultaria em ganhos satisfatórios, diminuindo o consumo dos recursos de hardware, os tempos de execução de *scripts* e consequentemente, melhorando a experiência do usuário.

Como proposta de trabalho futuro, uma sugestão seria o aperfeiçoamento da solução desenvolvida, implementando novas funcionalidades e construindo novos módulos. Dessa forma, provavelmente se teria um *software* consumindo mais recursos de *hardware*, possibilitando a realização de outros testes e a observação de outros parâmetros. Alguns exemplos de melhorias seriam:

a) Implementar outras operações sobre as imagens carregadas (mais filtros por exemplo);
b) Trabalhar também com animações;
c) Utilizar mais *web workers* (determinando em qual ponto eles deixariam de ser viáveis);
d) Utilizar outros *softwares* ou ferramentas de medição de desempenho, além das disponibilizadas pelos próprios navegadores.

Por fim, espera-se que este trabalho possa contribuir de alguma forma para aqueles que estão iniciando um estudo a respeito do paralelismo em linguagem *JavaScript* e que, de certa forma, incentive o desenvolvimento de outras pesquisas que contribuirão diretamente para o enriquecimento da área abordada.

REFERÊNCIAS

About the XAMPP Project. Apache Friends, 2016. Disponível em: <https://www.apachefriends.org/pt_br/about.html>. Acesso em: 24 de jan. de 2021.

ALECRIM, Emerson. **Processadores:** clock, bits, memória cache e múltiplos núcleos. InfoWester, 2008. Disponível em: <https://www.infowester.com/processadores.php#multicore>. Acesso em 03 ago. 2020.

AMOROSO, Danilo. **O que são processos de um sistema operacional e por que é importante saber.** TecMundo, 2009. Disponível em: <https://www.tecmundo.com.br/memoria/3197-o-que-sao-processos-de-um-sistema-operacional-e-por-que-e-importante-saber.htm>. Acesso em 15 set. 2020.

BARNEY, Blaise. **Introduction to Parallel Computing**. Lawrence Livermore National Laboratory, 2018. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em 23 ago. 2020.

BASQUES, Kayce. **Get Started With Analyzing Runtime Performance.** Chrome Dev Tools, 2020. Disponível em: <https://developers.google.com/web/tools/chrome-devtools/evaluate-performance>. Acesso em: 15 de jan. de 2021

BIDELMAN, Eric. **Conceitos básicos sobre serviços da web**. HTML5Rocks, 2010. Disponível em: <https://www.html5rocks.com/pt/tutorials/workers/basics/>. Acesso em 08 out. 2020.

Canvas Tutorial. Tecnologia Web para desenvolvedores, MDN Mozilla, 2019. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/Guide/HTML/Canvas_tutorial>. Acesso em: 09 de fev. de 2021.

Chromium. The Chromium Projects, 2008. Disponível em: <https://www.chromium.org/Home>. Acesso em: 10 de fev. de 2021.

CIPOLI, Pedro. **Quais são as vantagens de ter um processador com múltiplos núcleos?** Canaltech, 2012. Disponível em: <https://canaltech.com.br/hardware/quais-sao-as-vantagens-de-ter-um-processador-com-multiplos-nucleos-2891/>. Acesso em 07 ago. 2020.

Console.time(). Tecnologia Web para desenvolvedores, MDN Mozilla, 2019. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/console/time>. Acesso em: 07 de fev. de 2021.

CORREA, Jefferson dos Santos. **Estudo sobre linguagens de programação paralela e os conceitos chave de paralelismo em nível de programação.** Instituto Municipal do Ensino Superior de Assis, 2013. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1011330377.pdf>. Acesso em 27 ago. 2020.

DALFOVO, Michael Samir; LANA, Rogério Adilson; SILVEIRA, Amélia. Métodos quantitativos e qualitativos: um resgate teórico. **Revista Interdisciplinar Científica Aplicada**, Blumenau, v.2, n.4, p.01-13, Sem II. 2008.

DIANA, Juliana. **Pesquisa Quantitativa e Pesquisa Qualitativa**. Diferença, 2017. Disponível em: <https://www.diferenca.com/pesquisa-quantitativa-e-pesquisa-qualitativa/>. Acesso em 18 set. 2020.

DIONISIO, Edson José. **Introdução ao Visual Studio Code.** DevMedia, 2016. Disponível em: <https://www.devmedia.com.br/introducao-ao-visual-studio-code/34418>. Acesso em 13 set. 2020.

Grayscale to RGB Conversion. Tutorials Point, 2013. Disponível em: <https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm>. Acesso em 23 jan. 2021.

HIWARALE, Uday. **Parallel programming in JavaScript using Web Workers.** Medium, 2018. Disponível em: <https://medium.com/jspoint/achieving-parallelism-in-javascript-using-web-workers-8f921f2d26db>. Acesso em 08 out. 2020.

HTMLCanvasElement.getContext(). Tecnologia Web para desenvolvedores, MDN Mozilla, 2021. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/getContext>. Acesso em: 10 de fev. de 2021.

HTML Canvas Graphics. W3Schools, 2012. Disponível em: <https://www.w3schools.com/html/html5_canvas.asp>. Acesso em 15 jan. 2021.

HTML Web Workers API. W3Schools, 2012. Disponível em: <https://www.w3schools.com/html/html5_webworkers.asp>. Acesso em 06 out. 2020.

JÚNIOR, Manoel Oran Barros Coelho. **Programação Paralela**. DevMedia, 2011. Disponível em: <https://www.devmedia.com.br/programacao-paralela/21405>. Acesso em 23 ago. 2020.

LAKATOS, Eva Maria; MARCONI, Marina de Andrade. **Metodologia científica**. São Paulo: Atlas, 2000. Metodologia do trabalho científico, v. 6, 2010.

LOEWE, Érico Souza. **Programação paralela utilizando NodeJS.** Medium, 2019. Disponível em: <https://medium.com/cwi-software/programa%C3%A7%C3%A3o-paralela-utilizando-node-js-7efe7e8e9f39>. Acesso em 19 ago. 2020.

LOSSIO, Rodrigo. **O que é e para que serve o XAMPP?** 2019. Disponível em: <https://oraculoti.com.br/2019/07/01/o-que-e-e-para-que-serve-o-xampp/>. Acesso em: 24 de jan. de 2021.

MACDONALD, Matthew. **A Simple Introduction to Web Workers in JavaScript.** Medium, 2019. Disponível em: < https://medium.com/young-coder/a-simple-introduction-to-web-workers-in-javascript-b3504f9d9d1c>. Acesso em 04 out. 2020.

MARQUES, Rafael. **O que é HTML? Entenda de forma descomplicada.** Homehost, 2019. Disponível em: <https://www.homehost.com.br/blog/tutoriais/o-que-e-html/>. Acesso em 12 set. 2020.

O que é Browser ou Navegador? Webshare, 2018. Disponível em: <https://www.webshare.com.br/glossario/o-que-e-browser-ou-navegador/>. Acesso em: 23 de jan. de 2021.

O que é CPU? Canaltech, 2015. Disponível em: <https://canaltech.com.br/hardware/o-que-e-cpu/>. Acesso em 03 ago. 2020.

O que é Thread? Canaltech, 2014. Disponível em: <https://canaltech.com.br/produtos/o-que-e-thread/>. Acesso em 12 set. 2020.

SANTOS, Suzana de Siqueira. **Introdução à programação paralela e distribuída.** Instituto de Matemática e Estatística da USP, 2015. Disponível em: <https://www.ime.usp.br/~gold/cursos/2015/MAC5742/reports/ProgramacaoFuncional.pdf>. Acesso em 01 set. 2020.

SILVA, Dilma Menezes. **Introdução à Programação Concorrente para a Internet**. Departamento de Informática PUC RIO, 1999. Disponível em: <http://www.inf.puc-rio.br/~francis/Intro_Prog_Concor.pdf>. Acesso em 25 ago. 2020.

SILVA, Giancarlo. **O que é e como funciona a linguagem JavaScript?** Canaltech, 2015. Disponível em: <https://canaltech.com.br/internet/O-que-e-e-como-funciona-a-linguagem-JavaScript/>. Acesso em 27 ago. 2020.

TIOBE Index for January 2021. Tiobe, 2021. Disponível em:<https://www.tiobe.com/tiobe-index/>. Acesso em 20 jan. 2021.

WALTON, Philip. **Total Blocking Time (TBT)**. WebDev, 2019. Disponível em: <https://web.dev/tbt/?utm_source=devtools>. Acesso em: 13 de fev. de 2021.

Web Workers API. MDN Web Docs Mozilla, 2019. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/Web_Workers_API>. Acesso em 06 out. 2020.

Worker. MDN Web Docs Mozilla, 2021. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Worker/Worker>. Acesso em 23 fev. 2021.

Yosemite National Park Landscape. Gallery World, 2014. Disponível em: <https://gallery.world/photo-544865-yosemite-national-park-ssha-pejzazh>. Acesso em 20 fev. 2021.

ZAMPIERI, Gabriel. **O que é JavaScript.** Hostinger, 2019. Disponível em: <https://www.hostinger.com.br/tutoriais/o-que-e-javascript/>. Acesso em 27 set. 2020.