



UNIVERSIDADE FEDERAL DE SANTA MARIA  
CENTRO DE TECNOLOGIA  
DEPARTAMENTO DE LINGUAGENS E SISTEMAS DE COMPUTAÇÃO  
CURSO DE SISTEMAS DE INFORMAÇÃO  
DISCIPLINA DE PROGRAMAÇÃO PARALELA

## TRABALHO FINAL - TÉCNICAS DE PARALELIZAÇÃO UTILIZANDO JAVASCRIPT

Autora: Marinara Rübenich Fumagalli  
Professora: Andrea Schwertner Charão

Santa Maria, 03 de julho de 2018

## Sumário

<b>1 Introdução .....</b>	<b>2</b>
<b>2 Códigos .....</b>	<b>3</b>
2.1 Simulando MultiThreads através de Callbacks.....	3
2.2 Simulando MultiThreads através do módulo async para node.js.....	6
2.3 MultiThread através de Web Workers .....	8
2.4 MultiThread através da biblioteca Parallel.js.....	9
<b>3 Referências .....</b>	<b>11</b>

# 1 Introdução

Cada dia mais precisamos nos preocupar com desempenho e rapidez nos nossos códigos, principalmente quando se entra no mundo da internet e dos navegadores, não temos mais paciência para esperar as páginas carregarem ou para nossas requisições serem atendidas para então podermos prosseguir com nossa navegação na rede.

Falando em navegadores, existe uma linguagem de programação que é extremamente necessária e muito utilizada no desenvolvimento web: o JavaScript (nome popular) ou ECMAScript® (nome oficial). JavaScript é uma linguagem orientada a objetos e se encontra na terceira camada do desenvolvimento para Web. Segundo diversas fontes, basicamente existem 3 camadas: a primeira é o HTML (camada da informação), a segunda é o CSS (camada da formatação) e a terceira é o JavaScript (camada de comportamento), ou seja, ele manipula as outras camadas de forma a dar “vida” e interatividade a elas, cada evento (clique, parada de mouse, alerta através de pop-up, transição, movimentação, slider de imagens, menu dinâmico, etc.) é ele quem comanda e quanto mais interativo, mais atrativo ao usuário, muitas coisas só são possíveis graças à ele.

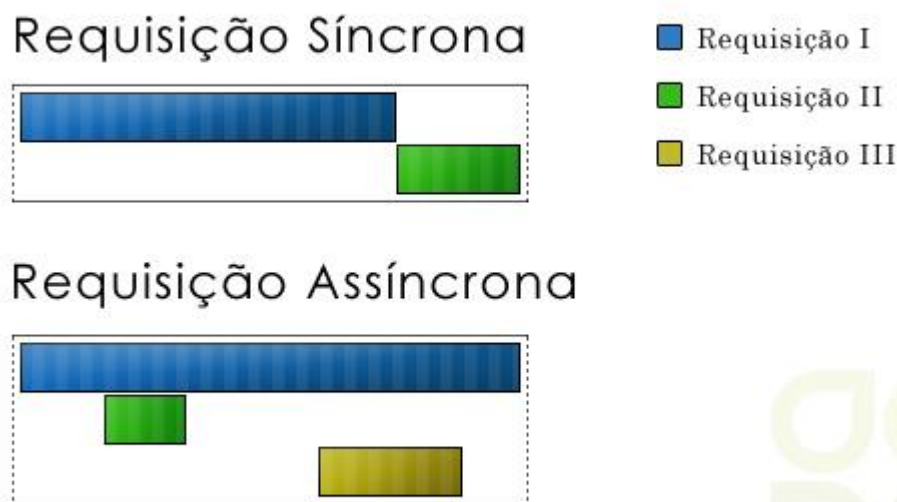
Porém, voltando a falar de desempenho, JS é uma linguagem de sequência única, ou seja, é utilizada apenas uma thread para qualquer função e dependendo da quantidade de informações que precisam ser carregadas ou manipuladas, o processo pode ser bastante lento. Então, como conseguir desempenho e rapidez com uma linguagem que trabalha com apenas uma thread? A chegada do HTML5 trouxe uma solução: os **Web Workers**, que são mecanismos que permitem que certas operações dos scripts possam ser executadas em segundo plano em threads diferentes da thread principal da aplicação, os Workers trocam mensagens entre eles para garantir o paralelismo, permitindo que tarefas sejam processadas sem que haja o bloqueio da thread principal, isso garante que não ocorram bloqueios, travamentos, erros e congelamento de tarefas, da tela ou até do navegador.

Neste relatório vou apresentar alguns exemplos de códigos que utilizam Workers através bibliotecas de paralelização e um em que é utilizada a criação e manipulação dos Workers em um só arquivo .js, em todos eles tem a comparação do tempo de execução sequencial x paralelo, também apresentarei imagens das saídas após a execução dos mesmos. Contudo, existem outras técnicas e inclusive esforços dos mantenedores da linguagem que fazem parte da **ECMA** (European Computer Manufacturers Association) para melhorar os Web Workers e encontrar outras soluções de paralelismo na programação para Web.

## 2 Códigos

Todos os códigos e Workers funcionam de forma **assíncrona**, ou seja, quando é feita uma requisição, as tarefas podem ser processadas paralelamente e não necessitam aguardar o término do processamento da outra para iniciarem. Então, o processo não precisa ficar bloqueado esperando uma resposta, do contrário, enquanto a resposta não chegasse nada mais poderia ser processado, o que nitidamente tornaria a aplicação lenta, congelaria a página e estaria bem propenso a gerar erros, tornando a experiência do usuário frustrante. A figura abaixo mostra um exemplo bem simples sobre as diferenças entre Requisição Síncrona e Requisição Assíncrona:

Fig. 1 - Requisição Síncrona e Assíncrona



Fonte: <http://www.diogomatheus.com.br/blog/php/requisicoes-sincronas-e-assincronas/>

### 2.1 Simulando MultiThreads através de Callbacks

Callback ocorre quando uma função é argumento de outra função e é executada quando se completa uma determinada tarefa. Neste primeiro exemplo será mostrado um código que é capaz de ler dois arquivos de texto quaisquer e imprimir o conteúdo dos dois arquivos no mesmo momento e se utilizará a função `console.time()` que permite fazer a contagem do tempo de execução para fins de comparação da execução normal x execução paralela. A primeira parte mostra a execução normal que aguarda o fim da leitura de todo o primeiro arquivo para então poder ser o segundo arquivo, o que causa uma demora da execução do mesmo.

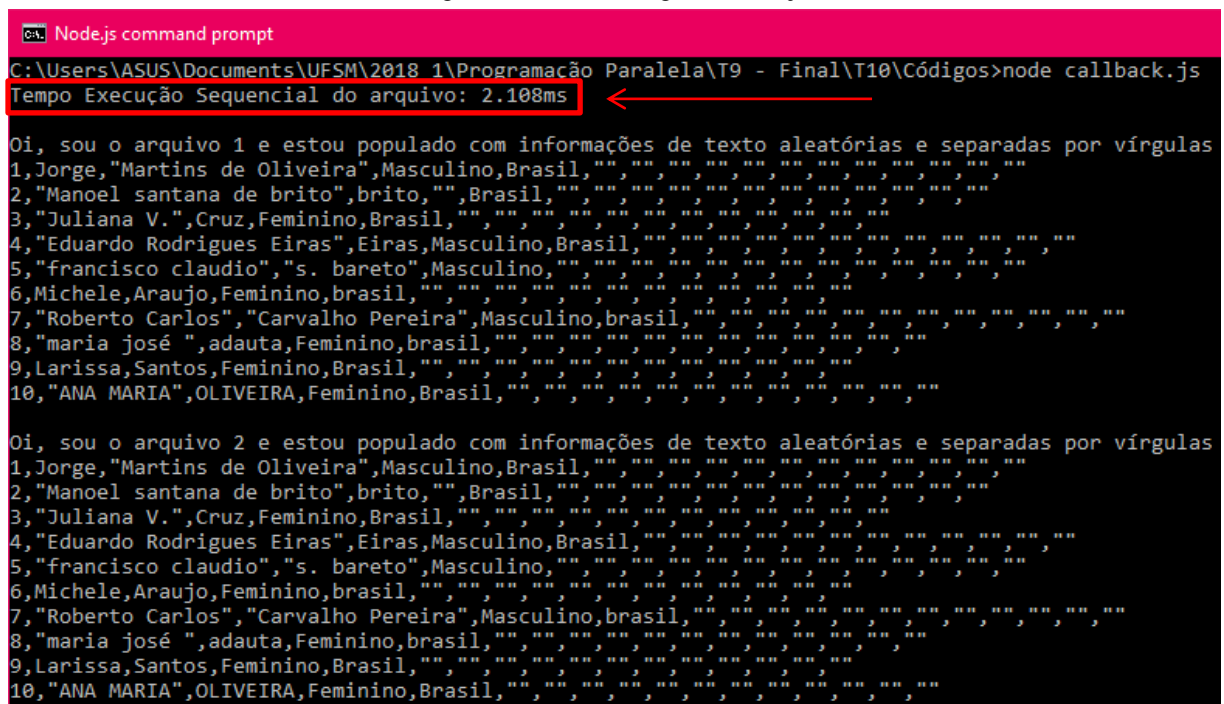
Fig. 2 – Código callback.js

```
1  const fs = require("fs");
2
3  console.time("Tempo Execução Sequencial do arquivo");
4  //Leitura do arquivo1.txt, se não ocorrer nenhum erro ele passa a ler o arquivo2
5  fs.readFile('./arquivo1.txt', function(err, arquivo1) {
6      if (err) {
7          return console.log(err);
8      }
9      else {
10         fs.readFile('./arquivo2.txt', function(err, arquivo2) {
11             if (err) {
12                 return console.log(err);
13             }
14             //Imprime os textos dos dois arquivos
15             console.log('\n' + arquivo1 + '\n\n' + arquivo2);
16         });
17     }
18     console.timeEnd("Tempo Execução Sequencial do arquivo");
19 });
```

Fonte: *print screen* do código criado através do editor Sublime Text

Para executá-lo foi utilizada a plataforma **Node.js**, com ele é possível criar aplicações web apenas com JavaScript e tudo ocorre de forma muito rápida e clara. Hoje em dia possui uma série de bibliotecas compatíveis, inclusive algumas que permitem utilizar mais de uma thread nos processamentos. Naturalmente ele utiliza apenas uma thread para processar as requisições de forma assíncrona, porém a biblioteca *parallel.js*, por exemplo, permite que se aproveitem os Web Workers para a Web e os processos filhos para o Node trazendo grande eficiência na paralelização. Também foi utilizado o gerenciador de pacotes para JavaScript: **npm**, para instalar bibliotecas e dependências necessárias. Por fim, a saída foi a seguinte:

Fig. 3 – Saída do Código callback.js



```
C:\Users\ASUS\Documents\UFSM\2018_1\Programação Paralela\T9 - Final\T10\Códigos>node callback.js
Tempo Execução Sequencial do arquivo: 2.108ms

Oi, sou o arquivo 1 e estou populado com informações de texto aleatórias e separadas por vírgulas
1,Jorge,"Martins de Oliveira",Masculino,Brasil,""
2,"Manoel santana de britto",britto,"",Brasil,""
3,"Juliana V.",Cruz,Feminino,Brasil,""
4,"Eduardo Rodrigues Eiras",Eiras,Masculino,Brasil,""
5,"francisco claudio","s. bareto",Masculino,""
6,Michele,Araujo,Feminino,brasil,""
7,"Roberto Carlos","Carvalho Pereira",Masculino,brasil,""
8,"maria josé ",adauta,Feminino,brasil,""
9,Larissa,Santos,Feminino,Brasil,""
10,"ANA MARIA",OLIVEIRA,Feminino,Brasil,""

Oi, sou o arquivo 2 e estou populado com informações de texto aleatórias e separadas por vírgulas
1,Jorge,"Martins de Oliveira",Masculino,Brasil,""
2,"Manoel santana de britto",britto,"",Brasil,""
3,"Juliana V.",Cruz,Feminino,Brasil,""
4,"Eduardo Rodrigues Eiras",Eiras,Masculino,Brasil,""
5,"francisco claudio","s. bareto",Masculino,""
6,Michele,Araujo,Feminino,brasil,""
7,"Roberto Carlos","Carvalho Pereira",Masculino,brasil,""
8,"maria josé ",adauta,Feminino,brasil,""
9,Larissa,Santos,Feminino,Brasil,""
10,"ANA MARIA",OLIVEIRA,Feminino,Brasil,""
```

Fonte: *print screen* da saída do código através do prompt do Node.js no SO Windows 10

Agora a segunda parte do código que lê os dois arquivos simultaneamente, guarda as informações deles até que ambas as leituras tenham sido concluídas:

Fig. 4 – Código callbackParalelo.js

```
1  const fs = require("fs");
2
3  var contador = 0;
4  //Variáveis que guardam o resultado de cada leitura, já que não se
5  //tem controle de qual leitura terminará primeiro
6  var leitura1, leitura2;
7  //O contador é incrementado a cada início de leitura e decrementado no fim,
8  //quando ele chegar a 0 significa que encerrou a leitura e pode imprimir
9  contador++;
10 console.time("Tempo Execução Paralela Total");
11 fs.readFile('./arquivo1.txt', function(err, dados) {
12     contador--;
13
14     if (err) {
15         return console.log(err);
16     }
17     else {
18         leitura1 = dados;
19
20         if (contador == 0) {
21             console.log('\n' + leitura1 + '\n\n' + leitura2);
22         }
23     }
24 });
25
26 contador++;
27 fs.readFile('./arquivo2.txt', function(err, dados) {
28     contador--;
29
30     if (err) {
31         return console.log(err);
32     }
33     else {
34         leitura2 = dados;
35
36         if (contador == 0) {
37             console.log('\n' + leitura1 + '\n\n' + leitura2);
38         }
39     }
40 });
41 console.timeEnd("Tempo Execução Paralela Total");
```

Fonte: *print screen* do código criado através do editor Sublime Text

Maiores explicações se encontram nas linhas comentadas na imagem do código (figura 4). E agora é apresentada a saída do código:

Fig. 5 – Saída do Código callbackParalelo.js

```
C:\> Node.js command prompt
C:\Users\ASUS\Documents\UFSM\2018_1\Programação Paralela\T9 - Final\T10\Códigos>node callbackParalelo.js
Tempo Execução Paralela Total: 0.675ms

Oi, sou o arquivo 1 e estou populado com informações de texto aleatórias e separadas por vírgulas
1,Jorge,"Martins de Oliveira",Masculino,Brasil,""
2,"Manoel santana de britto",britto,""
3,"Juliana V.",Cruz,Feminino,Brasil,""
4,"Eduardo Rodrigues Eiras",Eiras,Masculino,Brasil,""
5,"francisco claudio","s. bareto",Masculino,""
6,Michele,Araujo,Feminino,brasil,""
7,"Roberto Carlos","Carvalho Pereira",Masculino,brasil,""
8,"maria josé ",adauta,Feminino,brasil,""
9,Larissa,Santos,Feminino,Brasil,""
10,"ANA MARIA",OLIVEIRA,Feminino,Brasil,""

Oi, sou o arquivo 2 e estou populado com informações de texto aleatórias e separadas por vírgulas
1,Jorge,"Martins de Oliveira",Masculino,Brasil,""
2,"Manoel santana de britto",britto,""
3,"Juliana V.",Cruz,Feminino,Brasil,""
4,"Eduardo Rodrigues Eiras",Eiras,Masculino,Brasil,""
5,"francisco claudio","s. bareto",Masculino,""
6,Michele,Araujo,Feminino,brasil,""
7,"Roberto Carlos","Carvalho Pereira",Masculino,brasil,""
8,"maria josé ",adauta,Feminino,brasil,""
9,Larissa,Santos,Feminino,Brasil,""
10,"ANA MARIA",OLIVEIRA,Feminino,Brasil,""
```

Fonte: *print screen* da saída do código através do prompt do Node.js no SO Windows 10

Após 10 testes estes foram os melhores resultados, o código que realiza a leitura simultaneamente chegou a ter um desempenho quase 4x melhor.

## 2.2 Simulando MultiThreads através do módulo async para Node.js

Por fim, o último código utiliza o módulo async. Podemos instalar ele da mesma forma que o parallel.js:

```
npm install async --save
```

Ele basicamente automatiza as funções explicadas na seção acima através de requisições assíncronas e se mostra muito eficiente também:

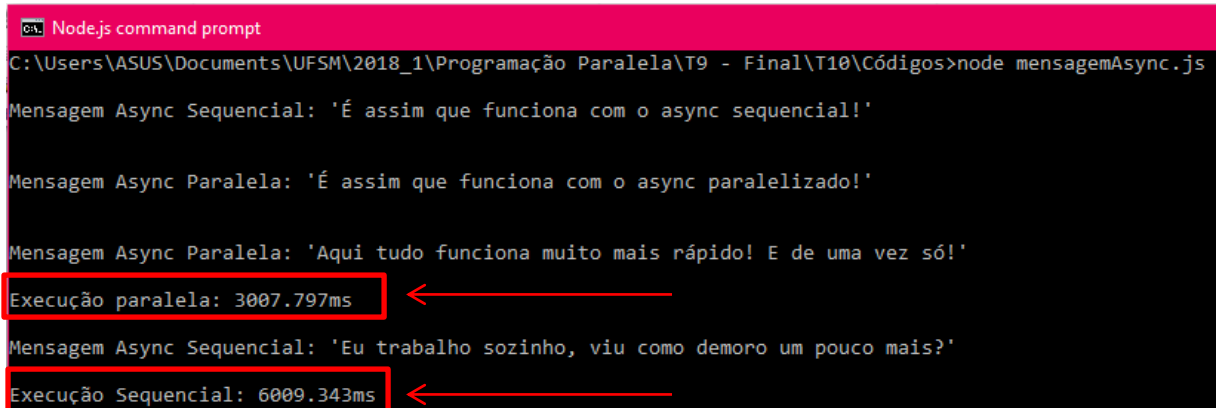
Fig. 6 – Código mensagemAsync.js

```
1  const async = require("async");
2
3  //Função que exibe no terminal uma mensagem (string) que é passada por parâmetro após passarem 3s
4  function exibeMensagem(mensagem, callback) {
5      setTimeout(function() {
6          console.log(mensagem);
7          callback();
8      }, 3000);
9  }
10
11 //Código Single - Ele recebe a primeira mensagem e imprime, então aguarda estar liberado para receber a segunda
12 console.time("Execução Sequencial");
13 exibeMensagem("\nMensagem Async Sequencial: 'É assim que funciona com o async sequencial!'\n", function() {
14     exibeMensagem("\nMensagem Async Sequencial: 'Eu trabalho sozinho, viu como demoro um pouco mais?'\n", function() {
15         console.timeEnd("Execução Sequencial");
16     });
17 });
18
19 //Código Paralelo - Neste caso ele pode receber e imprimir as duas mensagens ao mesmo tempo. Sem a necessidade de Callbacks aninhados
20 console.time("Execução paralela");
21 async.parallel([
22     function(callback) {
23         exibeMensagem("\nMensagem Async Paralela: 'É assim que funciona com o async paralelizado!'\n", function() {
24             callback();
25         });
26     },
27     function(callback) {
28         exibeMensagem("\nMensagem Async Paralela: 'Aqui tudo funciona muito mais rápido! E de uma vez só!'\n", function() {
29             callback();
30         });
31     }
32 ], function() {
33     console.timeEnd("Execução paralela");
34 });
35
```

Fonte: *print screen* do código criado através do editor Sublime Text

Eis a saída:

Fig. 7 – Saída do Código mensagemAsync.js



```
Node.js command prompt
C:\Users\ASUS\Documents\UFSM\2018_1\Programação Paralela\T9 - Final\T10\Códigos>node mensagemAsync.js

Mensagem Async Sequencial: 'É assim que funciona com o async sequencial!'

Mensagem Async Paralela: 'É assim que funciona com o async paralelizado!'

Mensagem Async Paralela: 'Aqui tudo funciona muito mais rápido! E de uma vez só!'

Execução paralela: 3007.797ms
Mensagem Async Sequencial: 'Eu trabalho sozinho, viu como demoro um pouco mais?'

Execução Sequencial: 6009.343ms
```

Fonte: *print screen* da saída do código através do prompt do Node.js no SO Windows 10

Também após 10 testes, estes foram os melhores resultados, o código que recebe e mostra a mensagem de forma assíncrona a ter um desempenho 2x melhor.



## 2.3 MultiThreads através de Web Workers

Na primeira parte deste arquivo forma criados manualmente os Workers que se comunicam e agem em segundo plano a fim de executarem o mais rápido possível, sem causar prejuízos às requisições e aos processos. Na segunda parte foram simuladas threads, ambos são incrementados no laço que vai de 0 até 1000000 e a execução é mostrada no navegador. Com o código em execução, nitidamente dá pra ver a diferença no tipo e no tempo de processamento. Mais informações encontram-se nos comentários no código, abaixo vão os códigos com o worker, a thread e a saída (estática):

Fig. 8 – Código laçoWorker.html – parte Workers

```
25 //Função que permite que o script trabalhe como script principal e worker ao mesmo tempo,
26 //sem a necessidade de dois arquivos.js
27 var criaWorker = function(foo){
28     var str = foo.toString().match(/^s*function\s*\(\s*\)\s*\{\{\{([\s\S](?!\\$))*[\s\S]\}\}\}[1];
29
30     return window.URL.createObjectURL(new Blob([str],{type: 'text/javascript'}));
31 }
32
33 //Web Worker: recebe um evento de uma String, enquanto o valor for menor que o passado por
34 //parâmetro: incrementa
35 var meuWorker = new Worker(criaWorker(function(){
36     self.addEventListener('message', function(e) {
37         var valor = 0;
38
39         while(valor <= e.data){
40             self.postMessage(valor);
41             valor++;
42         }
43     }, false);
44 }));
45 //Listener para o trabalhador, obtém os resultados da 1ª função e mostra na tela os resultados
46 meuWorker.addEventListener('message', function(e) {
47
48
49
50
51
52
53
54
55
56
57
58
59 //Enviamos o valor máximo aos workers e damos início a contagem
60 meuWorker.postMessage(100000);
61 meuWorker1.postMessage(100000);
62
```

Fonte: *print screen* do código criado através do editor Sublime Text

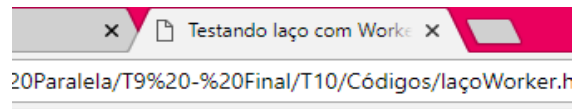
Fig. 9 – Código laçoWorker.html – parte Threads

```
64 //UTILIZAÇÃO DE SINGLE THREADS
65 function minhaThread(valor, valorMax){
66     var este = this;
67     document.getElementById("resThread").innerHTML = valor;
68     valor++;
69     //Utiliza recursividade até chegar ao valor máximo passado por parâmetro, ou seja,
70     //toda vez que o valor for menor que o valor passado ele se chama novamente
71     if(valor <= valorMax)
72         setTimeout(function () { este.minhaThread(valor, valorMax); }, 0);
73 }
74
75 minhaThread(0, 100000);
76 minhaThread1(0, 100000);
```

Fonte: *print screen* do código criado através do editor Sublime Text

E enfim, a saída:

Fig. 10 – Saída do Código laçoWorker.html



## Workers x Sequencial

### Workers: uso de MultiThreads para JavaScript

100000

100000

### Threads: simulando Workers para JavaScript

15501

15500

Fonte: *print screen* da saída do código através do navegador Google Chrome

## 2.4 MultiThreads através da biblioteca Parallel.js

Parallel.js, como já mencionado, é uma biblioteca que permite a computação paralela no JavaScript, ele separa as threads de processamento da thread principal. Possui muitas funções e está cada vez mais sendo usado por desenvolvedores web, mostrando ter um grande potencial na paralelização utilizando os workers.

Instalá-lo é bem simples, basta abrir o console do npm e colocar o comando:

```
npm install paralleljs --save
```

O código desta seção deve nos mostrar a soma de quadrados quaisquer, para isso chama o arquivo parallel.js no início do código realiza o paralelismo através da função map() que mapeia todos os números dados como parâmetro e os eleva ao quadrado. O código que contém também a função utilizando uma thread vem a seguir:

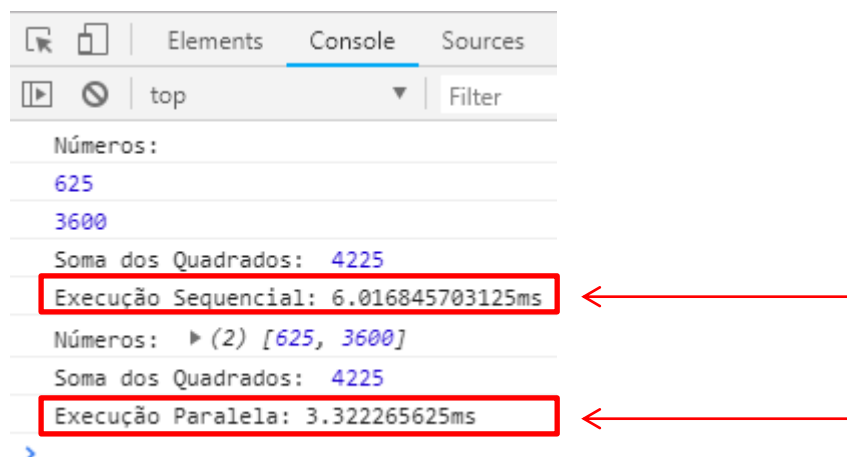
Fig. 11 – Código somaQuadrados.html

```
14 var elevaQuadradoP = {
15   //Retorna o quadrado dos números passados por parâmetro
16   pegaNumero: function(numero){
17     return Math.pow(numero,2);
18   }
19 }
20 var parallelObject = new Parallel([25,60]);
21
22 parallelObject.map(elevaQuadradoP.pegaNumero).
23   then(function(quadrado){
24     console.time("Execução Paralela");
25     console.log("Números: ", quadrado);
26     console.log("Soma dos Quadrados: ", quadrado[0] + quadrado[1]);
27     console.timeEnd("Execução Paralela");
28   });
29
30 //FUNÇÃO UTILIZANDO SINGLETHREADS
31 function elevaQuadradoT(x) {
32   return x * x;
33 }
34
35 function somaDosQuadrados(numeros) {
36   let soma = 0;
37
38   console.time("Execução Sequencial");
39   console.log("Números: ");
40   for (let i = 0; i < numeros.length; i++) {
41     soma += elevaQuadradoT(numeros[i]);
42     console.log(elevaQuadradoT(numeros[i]));
43   }
44   return(soma);
45 }
46 console.log("Soma dos Quadrados: ", somaDosQuadrados([25,60]));
47 console.timeEnd("Execução Sequencial");
```

Fonte: *print screen* do código criado através do editor Sublime Text

E a saída:

Fig.12 – Saída do Código somaQuadrados.html



Fonte: *print screen* da saída do código através do navegador da ferramenta de Debug do Google Chrome

Após alguns testes fica claro que a execução paralela se mostra mais eficiente, neste caso alcançou um SpeedUp 1,81.

### 3 Referências

1. Archibald, Jake. Promises em JavaScript: uma introdução. Disponível em: <<https://developers.google.com/web/fundamentals/primers/promises?hl=pt-br>>.
2. Assuncao, Charles. Web Worker: O jeito JS de fazer multithread. Disponível em: <<https://pt.linkedin.com/pulse/web-worker-o-jeito-js-de-fazer-multithread-charles-assuncao>>.
3. Async. Async Docs. Disponível em: <<https://caolan.github.io/async/docs.html>>.
4. Batista, Amós. Meu site, passo-a-passo— #4—Utilizando NPM para instalação de pacotes. Disponível em: <<https://medium.com/tableless/criando-o-meu-novo-site-4-utilizando-npm-para-instala%C3%A7%C3%A3o-de-pacotes-6c7cea2ab4b3>>.
5. Bidelman, Eric. Conceitos básicos sobre serviços da web. Disponível em: <<https://www.html5rocks.com/pt/tutorials/workers/basics/>>.
6. Emer, Jean Carlo. Fluxo de execução assíncrono em JavaScript – Callbacks. Disponível em: <<https://tableless.com.br/fluxo-de-execucao-assincrono-em-javascript-callbacks/>>.
7. Klemm, Justin. Node.js Async Tutorial. Disponível em: <<https://justinklemm.com/node-js-async-tutorial/>>.
8. MDN Web Docs. O que é JavaScript?. Disponível em: <[https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/First\\_steps/O\\_que\\_e\\_JavaScript](https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/First_steps/O_que_e_JavaScript)>.
9. MDN Web Docs. Using Web Workers. Disponível em: <[https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Workers\\_API/Using\\_web\\_workers](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers)>.
10. Node.js. Node.js Documentation. Disponível em: <<https://nodejs.org/dist/latest-v8.x/docs/api/documentation.html>>.
11. NPM. async-parallel. Disponível em: <<https://www.npmjs.com/package/async-parallel>>.
12. Peng, Max. Multithreading JavaScript. Disponível em: <<https://medium.com/techtrument/multithreading-javascript-46156179cf9a>>.
13. Santos, Guilherme. Node.js—O que é, por que usar e primeiros passos. Disponível em: <<https://medium.com/thdesenvolvedores/node-js-o-que-%C3%A9-por-que-usar-e-primeiros-passos-1118f771b889>>.
14. Tableless Padrões Web. O que é JavaScript? Controlando o comportamento do HTML e o CSS. <Disponível em: <http://tableless.github.io/iniciantes/manual/js/>>.
15. Welihinda, Amanda; Maxwell, Faustein; Rangel Mathias. Parallel.js - Easy multi-core processing with javascript. Disponível em: <<https://parallel.js.org/>>.
16. WHATWG community. Web Workers. Disponível em: <<https://html.spec.whatwg.org/multipage/workers.html>>.
17. Wendel, Erick. Gerenciando o fluxo assíncrono de operações em NodeJS. Disponível em: <<https://imasters.com.br/desenvolvimento/gerenciando-o-fluxo-assincrono-de-operacoes-em-nodejs>>.