Paralelizando um algoritmo de *backtracking* no navegador com Web Workers e WebAssembly

Daniel T. Rodrigues[1], **Calebe de Paula Bianchini**[1]

[1]Faculdade de Computação e Informática – Universidade Presbiteriana Mackenzie (UPM)
São Paulo, Brasil

danitrod@protonmail.com, calebe.bianchini@mackenzie.br

Resumo. *O uso de navegadores modernos se mostra cada vez mais essencial na atualidade. Recursos como Web Workers vêm se tornando mais adotados nos mais usados navegadores da Internet, possibilitando melhoria de desempenho em aplicações web, e por consequência, execução de tarefas de maior demanda computacional dentro desses mesmos navegadores. Este artigo explora uma técnica de paralelismo de tarefas usando Web Workers, apresentando como estudo de caso um algoritmo de geração de palavras cruzadas, executando-os em um navegador. Os resultados mostram, até mesmo, speedups superlineares para a versão paralela do algoritmo estudado.*

1. Introdução

O uso da Internet se torna maior a cada dia que passa, tendo crescido 1331.9% nas últimas duas décadas e atingido cerca de 5.17 bilhões de usuários, 65.6% da população mundial, em 2021 [Stats 2021]. Tarefas que comumente eram realizadas em aplicações *desktop* ou somente em servidores estão tendendo para os navegadores, evitando a necessidade de instalação de *softwares* especializados e diminuindo a sobrecarga em servidores. Havendo essa tendência de execução de tarefas de maior demanda computacional nos navegadores, se faz necessário estruturar melhor as aplicações que funcionam em navegadores.

Este artigo explora o modelo de paralelismo de tarefas que pode ser aplicado nos navegadores tradicionais. Para isso, foi escolhido um problema de geração de palavras cruzadas que é considerado NP-Completo [Garey and Johnson 1979]. Dessa forma, pode-se avaliar se os resultados alcançados apontam para uma maneira eficiente de solucionar o problema em uma página web.

2. Web Workers

No contexto de web, Workers são *scripts* (trechos de código) que executam em plano de fundo, em uma *thread* separada no navegador [WHATWG 2021b], de tal modo que uma tarefa sendo processada em um Worker não irá impactar, bloquear ou deixar a *thread* principal da página web mais lenta de alguma forma [Mozilla 2021]. Geralmente, é assumido que Workers têm um grande custo computacional para serem inicializados, e um alto custo de memória por instância. Por esse fato, Workers são geralmente usados para tarefas longas, em que o tempo de inicialização não será tão relevante em relação ao tempo de uso [W3C 2021]. Um Worker pode ser construído diretamente programado e invocado em JavaScript, sendo essa construção no *script* principal ou em outros Workers. Dentro de um Worker, pode-se utilizar qualquer biblioteca disponível de um navegador, exceto manipulações diretas do Modelo de Objeto de Documentos (DOM – *Document*

Object Model) [WHATWG 2021a], e alguns métodos específicos da página principal [Mozilla 2021].

3. WebAssembly

WebAssembly surgiu pela primeira vez oficialmente em 2017, desenvolvido em um grupo de comunidade da World Wide Web Consortium (W3C) com representantes de todos os principais navegadores web [WCG 2019], e foi fortemente influenciado pela tecnologia similar denominada asm.js [Halic 2019].

asm.js é um conjunto de funcionalidades do JavaScript publicado em 2013, designado para permitir que *software* escrito em outras linguagens, como C, possa ser executado dentro do navegador, uma vez compilado [Wagner 2013]. Com a execução de código em mais baixo nível que JavaScript, é possível obter melhor desempenho no navegador por abrir mão de algumas abstrações do JavaScript, como tipagem dinâmica e uso de coletor de lixo (*garbage collector*) [Herman et al. 2014] [Abelson et al. 1996].

WebAssembly aproveita parte das funcionalidades do *asm.js*, porém, não como um conjunto de funcionalidades executáveis dentro do JavaScript, mas como um novo formato de código binário intermediário que é diretamente executável no navegador [Rauschmayer 2015]. O uso de WebAssembly não impede o uso de JavaScript e vice-versa; ambos executam em conjunto no navegador. Estudos recentes [Herrera et al. 2018], com experimentos realizados nos navegadores Chrome 63, Firefox 57 e Safari 11, mostram que a execução de um algoritmo implementado em WebAssembly é mais rápida, em todos os navegadores, que a execução do mesmo algoritmo implementado em JavaScript.

4. Geração de Palavras Cruzadas

O algoritmo de Geração de Palavras Cruzadas escolhido tem como entrada uma codificação em formato de matriz das palavras cruzadas e um dicionário contendo as possíveis palavras que podem ser utilizadas no preenchimento da matriz. O algoritmo utiliza a técnica de *backtracking* para encontrar uma solução, sabendo que a principal restrição é satisfazer as letras em comum entre as palavras adjacentes e transversais. Essa busca é repetida para cada linha e coluna da matriz, procurando uma solução a partir da combinação das palavras disponíveis no dicionário. Esse processo é realizado até que uma solução seja encontrada ou todo o dicionário seja explorado sem sucesso.

Uma solução sequencial inicial foi desenvolvida utilizando uma aplicação web híbrida com uma interface gráfica em JavaScript e um gerador de palavras cruzadas escrito em Rust e compilável para WebAssembly. Essa solução faz uso de algumas heurísticas simples para a seleção das próximas palavras candidatas, de modo a reduzir a escolha de caminhos sem solução.

5. Experimento

A solução paralela adotada utiliza uma estratégia de divisão e conquista aplicada ao problema de *backtracking* [Rao and Kumar 1987]. Dessa forma, o trabalho de busca é dividido entre as *threads* disponíveis, de maneira equilibrada, e o algoritmo termina quando uma delas encontrar uma resposta, ou quando todas terminarem sem sucesso.

A implementação dessa solução[1] possui uma *thread* principal, que é responsável pela interface gráfica em JavaScript, e várias *threads* trabalhadoras, que instanciam a parte híbrida construída em Rust e compilada para WebAssembly. A *thread* principal criará um número definido de Web Workers, que são mapeados nas *threads* trabalhadoras. Cada *thread* trabalhadora recebe os dados necessários para a busca da solução, considerando que a divisão igualitária do dicionário entre as threads é o fator principal de garantia de balanceamento de carga entre elas. Essa divisão é sempre igual entre execuções diferentes, não havendo nenhum fator de aleatoriedade ou não-determinismo nela. A *thread* principal, após criar as trabalhadoras, fica esperando por eventos, até que alguma das *threads* encontre uma solução, ou todas elas terminem sem uma solução. Caso alguma das *threads* trabalhadoras encontre uma solução, a *thread* principal é notificada e também notifica as demais *threads* para terminarem o processo de busca.

Para a execução dos experimentos, foi utilizado um *notebook* macOS com processador Intel Core i9 2.3GHz (8c/16t) com 16GB de memória RAM. O navegador utilizado foi o Firefox 94. A Tabela 1 e a Figura 1 apresentam os tempos, em milissegundos, e os *speedups* obtidos na média de 10 execuções do experimento, respectivamente.

Número de threads	Sequencial	2	4	8	16
Média	8723	9184.5	5285.1	718.2	1287.9
Mediana	8671	9130	5323	716	1281.5
Desvio Padrão	218.56	215.47	118.29	30.62	21.01

Tabela 1. Tempos obtidos (em milissegundos).

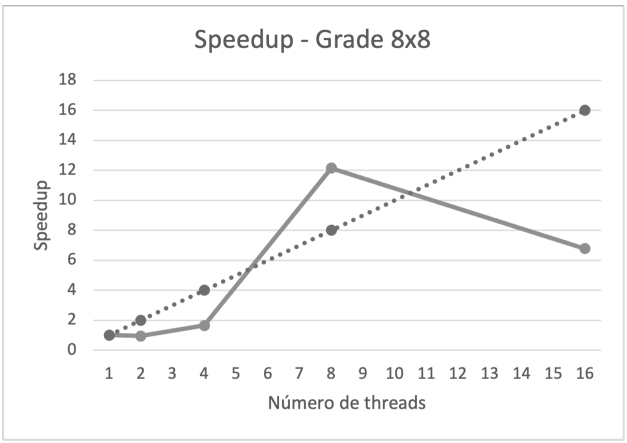

Figura 1. *Speedup* obtido.

É notável que a execução do algoritmo com duas *threads* aumentou levemente o tempo da solução, apesar do tempo diminuir com o uso de mais *threads*. A primeira explicação é que o custo de criação e inicialização de duas *threads* foi determinante para o aumento no tempo. A partir do uso de mais *threads*, a solução do problema está intimamente ligada à forma em como o dicionário é utilizado, pois uma palavra procurada pode estar em qualquer posição do dicionário, alterando, assim, o determinismo da solução. A probabilidade de encontrar um caminho melhor aumenta juntamente ao número de *threads*. Esse comportamento pode ser percebido no *speedup* obtido com quatro e oito

[1]Encontrada em `https://github.com/danitrod/crossword-composer`

threads. A partir de 16 *threads*, nota-se que o *speedup* diminuiu, e muito provavelmente com mais *threads* iria diminuir mais ainda, pois um caminho ótimo pode ter sido encontrado com menos *threads*, sendo então apenas acrescido o custo de inicialização dos Workers quando se acrescentam mais *threads*. Outro fato que contribui para a queda de *speedup* com 16 *threads* é que o processador utilizado tem somente 8 núcleos físicos.

6. Conclusões e Recomendações

Os resultados apresentados mostram que é possível utilizar computação paralela de forma eficaz dentro de navegadores, com a API Web Workers e o uso de uma linguagem rápida e segura como o Rust. O paralelismo do algoritmo apresentado mostrou um *speedup* superlinear nos melhores casos.

Como trabalhos futuros, pretende-se fazer uma comparação entre a execução da solução diretamente no sistema operacional versus no navegador. Para melhoria do algoritmo, pretende-se explorar outras heurísticas para seleção dos caminhos da árvore de busca, ou identificar caminhos que não devem ser percorridos. Pode-se também utilizar alguma análise probabilística de posições de letras no dicionário. Além disso, pode-se migrar a interface gráfica da aplicação de JavaScript para Rust/WebAssembly, criando uma base de código mais consistente e mais facilmente testável.

Referências

Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman & Co.

Halic, C. P. S. (2019). Estudo do processo de compilação para webassembly.

Herman, D., Wagner, L., and Zakai, A. (2014). asm.js, http://asmjs.org/spec/latest/.

Herrera, D., Chen, H., Lavoie, E., and Hendren, L. (2018). Webassembly and javascript challenge: Numerical program performance using modern browser technologies and devices. Technical report, University of McGill.

Mozilla (2021). Web workers api, https://developer.mozilla.org/en-us/docs/web/api/web_workers_api.

Rao, V. N. and Kumar, V. (1987). Parallel depth first search. part i. implementation. *International Journal of Parallel Programming*, 16:479–499.

Rauschmayer, A. (2015). Webassembly: a binary format for the web, https://2ality.com/2015/06/web-assembly.html.

Stats, I. W. (2021). Internet usage statistics, https://internetworldstats.com/stats.htm.

W3C (2021). Web workers. Technical report, World Wide Web Consortium.

Wagner, L. (2013). asm.js in firefox nightly, https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/.

WCG (2019). Webassembly roadmap, https://webassembly.org/roadmap/.

WHATWG (2021a). Dom living standard, https://dom.spec.whatwg.org/.

WHATWG (2021b). Html living standard, https://html.spec.whatwg.org/.