Performance im Web: Effektive Nutzung von WebAssembly in Kombination mit Multithreading

**Masterarbeit**  
Studiengang Master Informatik M.Sc  
der Hochschule Ruhr West

**Jonas Stratmann  
10009378**

Erstprüfer: Michael Schellenbach  
Zweitprüfer: Sven Tauhardt

Kooperationspartner: fluxatron softec GmbH  
Betreuer: Sven Tauhardt

Essen, April 2024

# 

Kurzfassung

Das Ziel dieser Arbeit ist sinnvoll.

Abstract

The aim of the present work is good.

Inhaltsverzeichnis

[Kurzfassung II](#_Toc165829596)

[Abstract II](#_Toc165829597)

[Inhaltsverzeichnis III](#_Toc165829598)

[Tabellenverzeichnis V](#_Toc165829599)

[Abbildungsverzeichnis VII](#_Toc165829600)

[Vorwort IX](#_Toc165829601)

[1 Einführung 1](#_Toc165829602)

[1.1 Motivation 2](#_Toc165829603)

[1.2 Ziel der Arbeit 2](#_Toc165829604)

[1.3 Methodik 3](#_Toc165829605)

[1.4 Gliederung der Arbeit 3](#_Toc165829606)

[2 Grundlagen 4](#_Toc165829607)

[2.1 JavaScript – Anfänge einer interpretierten Sprache für die Webentwicklung 4](#_Toc165829608)

[2.2 Merkmale und Funktionsweise von JavaScript 5](#_Toc165829609)

[2.2.1 Die JavaScript Engine 6](#_Toc165829610)

[2.3 WebAssembly – ein optimiertes Format für die JavaScript Engine 10](#_Toc165829611)

[2.4 Multithreading 13](#_Toc165829612)

[2.4.1 Speichernutzung in Threads 14](#_Toc165829613)

[2.4.2 Multithreading in JavaScript und Webassembly 15](#_Toc165829614)

[2.4.3 Javascripts Konkurrenzmodelle 16](#_Toc165829615)

[2.4.4 Thread Pooling 17](#_Toc165829616)

[3 Stand der Technik 18](#_Toc165829617)

[3.1 Anwendungsbeispiele von WebAssembly in der Industrie 18](#_Toc165829618)

[3.1.1 Ebay 19](#_Toc165829619)

[3.1.2 TensorFlow.js 20](#_Toc165829620)

[3.1.3 FFmpeg 21](#_Toc165829621)

[3.2 Verwandte Arbeiten 23](#_Toc165829622)

[3.2.1 WebAssembly Performance 23](#_Toc165829623)

[3.2.2 Bounds-Checking 26](#_Toc165829624)

[3.2.3 Implementierungsansätze 26](#_Toc165829625)

[3.2.4 Analyse der Multithreading-Strategien 28](#_Toc165829626)

[4 Methodik 30](#_Toc165829627)

[4.1 Implementierungsansätze 30](#_Toc165829628)

[4.2 Das gewählte Leistungsproblem 32](#_Toc165829629)

[4.2.1 Kategorisierung der Leistungsprobleme 33](#_Toc165829630)

[4.2.2 Integration der Leibniz-Reihe zur Steuerung der Rechenintensität 33](#_Toc165829631)

[4.3 Parameterkonfigurationen 33](#_Toc165829632)

[4.4 Erhobene Performance Metriken 34](#_Toc165829633)

[4.5 Erfassung der Daten 35](#_Toc165829634)

[4.6 Abgrenzung 35](#_Toc165829635)

[5 Architektur 38](#_Toc165829636)

[5.1 wasm\_sharedMemory\_pthreads 38](#_Toc165829637)

[5.2 wasm\_sharedMemory\_webWorkers 39](#_Toc165829638)

[5.3 wasm\_actor\_webWorker 40](#_Toc165829639)

[5.4 wasm\_actor\_pthreads 42](#_Toc165829640)

[6 Implementierung 43](#_Toc165829641)

[6.1 Test-Suite 43](#_Toc165829642)

[6.2 Leistungsprobleme 48](#_Toc165829643)

[6.3 Implementierungsansätze 53](#_Toc165829644)

[6.3.1 wasm\_sharedMemory\_pthreads 55](#_Toc165829645)

[6.3.2 wasm\_sharedMemory\_webWorkers 60](#_Toc165829646)

[6.3.3 wasm\_actor\_webWorkers 63](#_Toc165829647)

[7 Ergebnisse 66](#_Toc165829648)

[8 Diskussion 67](#_Toc165829649)

[9 Ausblick 68](#_Toc165829650)

[Anhang A Abbildungen 69](#_Toc165829651)

[Anhang B Quellcode: 70](#_Toc165829652)

[Literaturverzeichnis 77](#_Toc165829653)

[Erklärung 82](#_Toc165829654)

Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

Abbildungsverzeichnis

[Abbildung 1: Most used programming languages among developers worldwide as 2023 (statista, 2023) 5](#_Toc165646487)

[Abbildung 2: Zusammenspiel von Event Loop, Web-API und JS-Engine (Traversy Media, 2022) 9](#_Toc165646488)

[Abbildung 3: Speichermanagement in JavaScript (Onwuemene, kein Datum) 10](#_Toc165646489)

[Abbildung 4: Unterschiede in der Kompilierung zwischen JavaScript im Vergleich zu WebAssembly (Gallant, Meet WebAssembly, 2019) 11](#_Toc165646490)

[Abbildung 5: WebAssembly im Browser, schematische Darstellung (Schnarr, 2023) 12](#_Toc165646491)

[Abbildung 6: Multithreading und gleichzeitige Verarbeitung mehrerer Threads (Mihaylov, 2019) 13](#_Toc165646492)

[Abbildung 7: Multithreading in einem Prozess mit geteiltem Speicher und individuellen Call-Stacks (Mihaylov, 2019) 14](#_Toc165646493)

[Abbildung 8: Entwicklerbeitrag zur Problematik der gemeinsamen Speichernutzung durch Threads (Batchelder, 2012) 14](#_Toc165646494)

[Abbildung 9: Thread-Kommunikation in JavaScript durch postMessage, wobei Objekte geklont statt geteilt werden (Mihaylov, 2019) 15](#_Toc165646495)

[Abbildung 10: Anwendungsarchitektur mit Hauptthread und drei Web-Workern, die unabhängige Module laden (Padmanabhan & Jha, 2019) 20](#_Toc165646496)

[Abbildung 11: Benchmark-Tests mit Leistungsgewinn durch 'Wasm + SIMD + Threads' (Yuan & Dukhan, 2020) 21](#_Toc165646497)

[Abbildung 12: Vergleich von Verarbeitungszeiten zwischen Single-Threaded 'Core v0.12.3', Multithreading 'Core-MT v0.12.3' und einem schnelleren nativen C-Build (FFmpeg, kein Datum) 22](#_Toc165646498)

[Abbildung 13: Architektur von FFmpeg-Multithreading mit Hauptthread und Worker-Kommunikation (FFmpeg, kein Datum) 23](#_Toc165646499)

[Abbildung 14: Modelle zur Interaktion von Nebenläufigkeit und Speicher (eigene Darstellung) 31](#_Toc165646500)

[Abbildung 15: WebAssembly Multithreading-Konzepte (eigene Darstellung) 31](#_Toc165646501)

[Abbildung 16: Architektur der Implementierung wasm\_sharedMemory\_pthread (eigene Darstellung) 39](#_Toc165646502)

[Abbildung 17: Architektur der Implementierung wasm\_sharedMemory\_webWorkers (eigene Darstellung) 40](#_Toc165646503)

[Abbildung 18: Architektur der Implementierung wasm\_actor\_webWorker (eigene Darstellung) 41](#_Toc165646504)

[Abbildung 19: Architektur der Implementierung wasm\_actor\_pthread (eigene Darstellung) 42](#_Toc165646505)

[Abbildung 20: Startseite der Test-Suite 44](#_Toc165646506)

[Abbildung 21: Ansicht eines Performance-Tests mit Graphenvisualisierung 45](#_Toc165646507)

[Abbildung 22: Aktivitätsdiagramm zur Visualisierung der implementierten DSatur-Funktion (eigene Darstellung) 49](#_Toc165646508)

Vorwort

Diese Masterarbeit verfasste ich als Abschlussarbeit meines Studiums der Angewandten Informatik an der Hochschule Ruhr West im Rahmen meiner Tätigkeit im Unternehmen fluxatron Softec GmbH. Das Unternehmen beschäftigt sich mit Verwaltungssoftware im Bereich der Energiewirtschaft.

TODO

Zur besseren Lesbarkeit wird in der vorliegenden Arbeit auf die gleichzeitige Verwendung männlicher und weiblicher Sprechform verzichtet. Es wird das generische Maskulinum verwendet, wobei beide Geschlechter gleichermaßen gemeint sind.

Ich wünsche Ihnen viel Freunde beim Lesen dieser Bachelorarbeit.

Jonas Stratmann

Essen, 20.04.2024

# Einführung

***„Performance is a feature“ – Steve Souders***

Mit diesem Leitsatz unterstrich Steve Souders - eine führende Autorität im Bereich der Web-Performance - die fundamentale Bedeutung der Geschwindigkeit und Effizienz von Webanwendungen. Performance sollte nicht als ein gegebenes Merkmal einer guten Software-Architektur betrachtet werden, sondern vielmehr als ein aktiv zu entwickelndes Feature. Diese Erkenntnis ist nicht nur ein Aufruf zur Optimierung, sondern bildet das Fundament für eine neue Ära der Webentwicklung, in der die Reaktionsgeschwindigkeit einer Webseite als entscheidender Faktor für den Erfolg im digitalen Raum gilt (Souders, 2007). Die Tragweite dieser Aussage findet ihre Bestätigung in den Erwartungshaltungen und Erfahrungen der Nutzer: laut einem Bericht von Google gaben 79 % der Webnutzer an, dass sie mit hoher Wahrscheinlichkeit keine Wiederholungskäufe auf einer Webseite tätigen werden, die unter schlechter Performance leidet (An & Meenan, 2016).

Die Relevanz von Hochleistungs-Webanwendungen erfährt eine zusätzliche Dimension, wenn man die Entwicklung des Webs betrachtet. Was einst als einfaches Netzwerk für den Austausch von Dokumenten begann (Berners-Lee, 1990), hat sich mittlerweile zur universellen Anwendungsplattform entwickelt, die heute über eine beeindruckende Bandbreite von Betriebssystemen und Gerätetypen hinweg zugänglich ist. Dies macht das Web zu einer zentralen Schnittstelle für eine Vielzahl von Anwendungen und Diensten – von komplexen Unternehmensanwendungen über soziale Medien bis hin zu anspruchsvollen interaktiven Spielen bietet das Web eine Plattform für nahezu unbegrenzte Möglichkeiten (Taivalsaari & Mikkonen, 2011).

Ein wesentlicher Engpass in der Performance von Webanwendungen ist jedoch, dass Webbrowser ausschließlich eine Programmiersprache unterstützen: JavaScript (Flanagan, 2020). Zwar bietet JavaScript eine Reihe von Vorteilen und Flexibilität, doch werden von ihm mittlerweile Leistungen abverlangt, für die es ursprünglich nicht konzipiert wurde. Dies umfasst rechenintensive Algorithmen, anspruchsvolle grafische Darstellungen und die schnelle Verarbeitung großer Datenmengen (Rauschmayer, 2014).

Nicht nur die Anforderungen an JavaScript haben sich im Laufe der Zeit gewandelt, sondern auch die Technologien, die uns zur Verfügung stehen, um diesen Anforderungen gerecht zu werden. Insbesondere im Hinblick auf die Performance sind die technologischen Möglichkeiten, mehrere Threads in JavaScript zu nutzen (Bewersdorff, 2014) und maschinennahen Code im Browser auszuführen (Wingler, 2018), von entscheidender Bedeutung. Diese Entwicklungen öffnen neue Türen, um die Web-Performance zu optimieren und die Effizienz im Umgang mit Ressourcen zu steigern.

In der vorliegenden Arbeit wird untersucht, wie diese modernen Technologien dazu beitragen können, die Vision von Souders umzusetzen, indem sie neue Wege eröffnen, den gestiegenen Anforderungen an Performance und Ressourceneffizienz von Webanwendungen gerecht zu werden. Durch die gezielte Nutzung dieser Technologien streben wir danach, die Grenzen der aktuellen Webentwicklung zu erweitern und die interaktive Geschwindigkeit von Webanwendungen signifikant zu verbessern und gleichzeitig einen optimalen, transparenten Ansatz für die Nutzung dieser Schlüsseltechnologien in ihrem Zusammenspiel zu demonstrieren.

## Motivation

In meiner Rolle als Webentwickler bei fluxatron softec konzentriere ich mich auf die Erstellung von Webanwendungen, die eine hohe Leistungsfähigkeit erfordern, um auf unterschiedlichsten Geräten der Kunden effizient zu funktionieren. Ein zentrales Element dieser Anwendungen ist ein innovatives Grafiktool, das speziell für die dynamische Visualisierung von Graphen entwickelt wurde. Diese Anwendungen mit ihren Funktionalitäten auf beliebigen Geräten performant auszuführen, stellt eine große Herausforderung dar, da insbesondere auf Graphen basierte Operationen und deren Visualisierung enorm rechenintensiv sein können.

Graphen können für eine Vielzahl von Anwendungen genutzt werden. Ihre Komplexität und die damit verbundenen Berechnungsprobleme sind Gegenstand intensiver Forschung in der Komplexitätstheorie, die diverse Problemstellungen im Zusammenhang mit Graphen definiert hat. Diese Probleme sind in nichtdeterministischer polynomieller Zeit (NP) lösbar, was sie aufgrund der potentiell hohen Leistungsanforderung für Webanwendungen besonders herausfordernd macht (Schöning, 1997).

Ein besonders interessantes NP-Problem in unserem Arbeitsumfeld ist das der Knotenfärbung. Hierbei geht es darum, die Knoten eines Graphen so einzufärben, dass keine zwei benachbarten Knoten dieselbe Farbe aufweisen. Dieses Problem findet praktische Anwendung in Bereichen wie Scheduling, Ressourcenzuweisung und Netzwerkdesign. Ein Anwendungsbeispiel hierfür ist die Planung und Optimierung von Netzwerken - etwa Transportnetzwerken - um Konflikte oder Überlastungen zu vermeiden.

Angesichts der Herausforderungen, die sich aus der Rechenintensität dieses Problems ergeben, haben wir uns bei fluxatron softec der Technologie WebAssembly zugewandt. Diese ermöglicht es uns, Code, der näher an der Maschinenebene operiert und beispielsweise in C++ geschrieben ist, direkt im Webbrowser auszuführen (Wingler, 2018). Dies stellt eine deutliche Verbesserung gegenüber traditionellem, interpretiertem JavaScript-Code dar. Außerdem lässt sich der für die Knotenfärbung genutzte Algorithmus hervorragend für die parallele Verarbeitung in verschiedenen Threads optimieren, um die Leistungsfähigkeit weiter zu steigern.

## Ziel der Arbeit

Bei der Implementierung der Performanceoptimierung für die Knotenfärbung in der Web-Visualisierungssoftware ergaben sich jedoch Fragen, insbesondere in Bezug auf die Parallelverarbeitung: Sollte diese auf der Ebene des WebAssembly Codes oder durch JavaScript erfolgen? Zudem standen wir vor der Herausforderung, die Kommunikation zwischen den Prozessen und den Zugriff auf die grafischen Ressourcen des Browsers für die Knotenfärbung zu organisieren.

Obwohl die Forschung bereits Ansätze zur Kombination von WebAssembly mit Multithreading bietet (Milner, 2019) (Stepanyan, kein Datum), fehlt es an klaren Richtlinien für das optimale Vorgehen. Das Ziel dieser Arbeit ist es, durch eine detaillierte Analyse dieser Technologien, eine klare und effiziente Methode für ihre Anwendung zu entwickeln, die nicht nur die Performance unseres Grafiktools verbessert, sondern auch als Leitfaden für die Optimierung von rechenintensiven Webanwendungen allgemein dienen kann. Vor diesem Hintergrund und der erkennbaren Notwendigkeit einer präzisen Methodik lässt sich die zentrale Forschungsfrage dieser Arbeit wie folgt definieren:

**Welche Ansätze und Techniken ermöglichen die effizienteste Nutzung paralleler Verarbeitung in Verbindung mit WebAssembly zur Maximierung der Web-Performance?**

TODO Erwartete Ergebnisse, geplantes System.

## Methodik

Herangehensweise – Performance -Ressourcen Vergleich der umgesetzten Systeme.

Beschreibe genauer, welche Forschungsmethoden du anwendest (z.B. vergleichende Analyse, Fallstudien, experimentelle Untersuchungen) und warum diese Methoden gewählt wurden.

## Gliederung der Arbeit

Der rote Faden.

# Grundlagen

In diesem Kapitel widmen wir uns den grundlegenden Konzepten, deren Verständnis essenziell ist, um die Herausforderungen zu verstehen, die sich aus der parallelen Verarbeitung in Kombination mit WebAssembly ergeben. Wir beginnen mit einem Überblick über ***JavaScript*** – die vorherrschende interpretierte Sprache im Web und die Besonderheiten ihrer Ausführungsumgebung, der ***JavaScript-Engine***. Anschließend werden wir das Format ***WebAssembly*** und seine Besonderheiten näher beleuchten. Abschließend rücken wir die Technologie hinter ***Multithreading*** - die parallele Ausführungsprozesse ermöglicht - in den Fokus unserer Betrachtung.

## JavaScript – Anfänge einer interpretierten Sprache für die Webentwicklung

Die Geschichte von JavaScript beginnt mit der Einführung des ersten Browsers, Mosaic, im Jahr 1993. Mosaic stellte eine Revolution dar, da er als erster Browser Text und Bilder gemeinsam darstellen konnte, wodurch das Internet einer breiten Öffentlichkeit ohne spezifische technische Kenntnisse zugänglich gemacht wurde. Diese Zugänglichkeit wurde durch eine interaktive und grafische Benutzeroberfläche ermöglicht (Borchers, 2003).

In dieser Anfangsphase diente das Internet hauptsächlich als einfaches Netzwerk zum Austausch von Dokumenten, wobei die erstellten Webseiten ausschließlich aus HTML-Elementen bestanden und somit statisch waren (Berners-Lee, 1990). Entwickler suchten nach Möglichkeiten, die statischen Seiten lebendiger zu gestalten, indem sie HTML-Elemente manipulieren und damit interaktive Nutzererfahrungen ermöglichen wollten.

Um diese Ambitionen zu realisieren, entwickelten sie den Mosaic-Browser weiter und führten Netscape ein, einen fortschrittlicheren Browser, der durch eine eingebaute Software eine neu entwickelte Programmiersprache namens "Mocha" interpretieren konnte. Diese Innovation erlaubte es Entwicklern, neben ihren HTML-Dateien auch Skripte zu integrieren, die dann auf Nutzerseite HTML-Elemente manipulieren, einfache Algorithmen ausführen und Berechnungen im Browser durchführen konnten (Goldberg & Wagner, 1996).

Mit der Zeit entwickelte sich die Sprache stetig weiter, unterstützt durch zahlreiche Features und Bibliotheken. Aus dem ursprünglichen Namen "Mocha" wurde zunächst "LiveScript" und schließlich "JavaScript". Diese Entwicklung markiert nicht nur die Evolution einer Programmiersprache, sondern auch einen entscheidenden Wendepunkt in der Geschichte des Internets, der die Weichen für dynamische und interaktive Webseiten stellte (Flanagan, 2020).

## Merkmale und Funktionsweise von JavaScript

Wie Abbildung 1 zeigt, hat sich JavaScript zur bevorzugten Programmiersprache von Softwareentwicklern weltweit entwickelt (Vailshery, 2024). Der herausragende Erfolg von JavaScript lässt sich auf seine einzigartigen Eigenschaften zurückführen. Als eine interpretierte Sprache, die von einer leichtgewichtigen Engine verarbeitet wird, zeichnet sich JavaScript durch seine Plattformunabhängigkeit aus. Es kann auf fast jedem Gerät ausgeführt werden, das über einen Webbrowser verfügt (Flanagan, 2020).

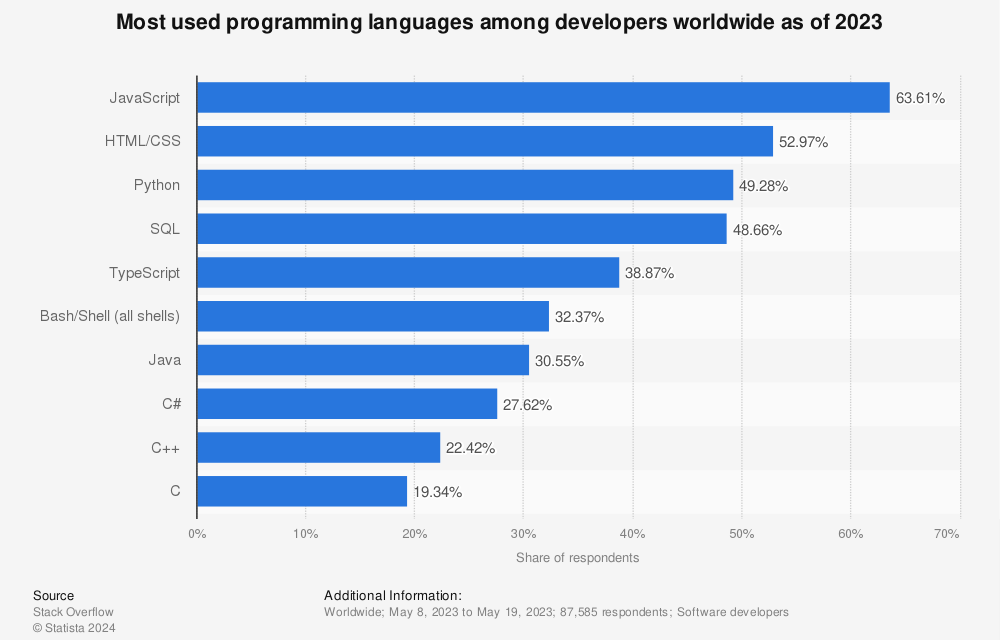


Abbildung 1: Most used programming languages among developers worldwide as 2023 (statista, 2023)

Im Gegensatz zu kompilierten Sprachen, bei denen der Code vor der Ausführung in Maschinencode für einen spezifischen Prozessortyp umgewandelt wird, interpretiert die Engine von JavaScript den Code zur Laufzeit Zeile für Zeile. Dabei wird jede Anweisung einzeln in den Maschinencode des jeweiligen Systems übersetzt. Diese Dynamik, die auch bei anderen interpretierten Sprachen vorkommt, ermöglicht es JavaScript, auf einer Vielzahl von Geräten und Betriebssystemen ohne spezifische Voranpassungen zu funktionieren (Flanagan, 2020).

Ein Nachteil der Interpretation zur Laufzeit ist, dass interpretierten Sprachen wie JavaScript im Vergleich zu kompilierten Sprachen tendenziell langsamer ist, da keine Optimierungen für einen bestimmten Prozessor im Voraus durchgeführt werden können (Flanagan, 2020).

Ein weiterer Vorteil dieser Flexibilität ist die dynamische Typisierung von JavaScript. Variablentypen werden erst zur Laufzeit bestimmt, was bedeutet, dass Variablen während unterschiedlicher Ausführungen verschiedene Typen annehmen können. Diese Eigenschaft fördert die Entwicklung dynamischer Anwendungen, da sie Entwicklern erlaubt, mit weniger strikten Typisierungsregeln zu arbeiten und sich auf die Logik und Funktionalität der Anwendung zu konzentrieren. Viele Entwickler betrachten die dynamische Typisierung von JavaScript als Nachteil, da eine statische Typisierung oft zu besser wartbarem Code führt. Allerdings bietet das Superset TypeScript eine Lösung. Es erweitert JavaScript, indem es die Möglichkeit bietet, typisierten Code zu schreiben. Dieser kann anschließend in JavaScript umgewandelt werden, wodurch die Vorteile beider Ansätze – der strengeren Typisierung und der Flexibilität von JavaScript – vereint werden. Als Superset fügt TypeScript der Basissprache zusätzliche Sprachattribute hinzu, die eine strengere Typisierung ermöglichen (Rosenwasser, 2022).

Analog zu vielen anderen Sprachen schöpft JavaScript seine grundlegende Syntax aus C, inklusive geschweifter Klammern, Semikolons und vordefinierter Schlüsselwörter. Das Ziel war die Schaffung einer leichteren, benutzerfreundlicheren Version von C, mit vereinfachten semantischen Strukturen und verbesserten Merkmalen für den Umgang mit dynamischem Speicher. Die begrenzte Lebensdauer von Webseiten, die nur einige Sekunden bis Minuten beträgt, erlaubte den ursprünglichen JavaScript-Versionen, Nebenläufigkeit und Speicherverwaltung in einer sehr vereinfachten Form anzugehen (Serverance, 2012).

Viele der zuvor genannten Features haben sich im Laufe der Zeit weiterentwickelt, um den steigenden Anforderungen an die Sprache gerecht zu werden. Beispielsweise hat die Zunahme der Nutzungsdauer in einer Session von Webanwendungen zu einem Bedarf an effizienterer Speicherverwaltung und Leistungsverbesserungen geführt. Diese Herausforderungen wurden durch die Weiterentwicklung von JavaScript-Engines und deren Schnittstellen zur Außenwelt angegangen, ein Thema, das wir im folgenden Abschnitt ausführlicher betrachten werden.

### Die JavaScript Engine

Wenn wir die Performance paralleler Verarbeitungsmethoden untersuchen und ermitteln möchten, welche Implementierungsweise in Kombination mit WebAssembly am leistungsfähigsten ist, ist es essenziell, das Verarbeitungsprinzip von JavaScript-Code zu verstehen. Hier spielt die JavaScript-Engine eine zentrale Rolle. Die Diskussion über JavaScript-Engines gestaltet sich allerdings komplex, da verschiedene Browser eigene Engines nutzen, die jeweils spezifische Vor- und Nachteile aufweisen. Die Differenzen zwischen den verschiedenen Engines resultieren hauptsächlich aus unterschiedlichen Ansätzen der Codeoptimierung nach dessen Interpretation. Trotz dieser Unterschiede können wir einen Überblick über die grundlegenden Paradigmen bieten, die sich in allen Engines gleichermaßen zeigen (CRYREX Enterprise, 2023).

Aktuell am häufigsten genutzte JavaScript-Engines:

* Mozilla Firefox: SpiderMonkey – die erste jemals erstellte Engine. Begann als rudimentäre Engine und erzeugt heute hochgradig optimierten Maschinencode.
* Google Chrome: V8 Engine – schnell und kann eigenständig betrieben werden. Die Node.js-Laufzeitumgebung nutzt die V8 Engine.
* Microsoft Edge: Chakra – eine proprietäre Engine von Microsoft.
* Safari: JavaScriptCore – wird von Safari sowie vielen anderen Apple-Anwendungen verwendet.

Gemäß aktuellen Benchmark-Tests ist die V8 Engine derzeit die schnellste, wobei die Performance je nach Art des Codes variieren kann. Bestimmte Engines können spezifische Muster bzw. Aufgaben besser optimieren (Woo, 2016) (Pitchkites, 2024).

#### Der Compiler

Im vorherigen Abschnitt erläuterten wir bereits, dass die JavaScript-Engine eine Software darstellt, die als Vermittler zwischen dem von Entwicklern verfassten JavaScript-Code und dem von Computern interpretierten Maschinencode fungiert, indem sie den Code zeilenweise interpretiert. Obwohl dies weiterhin das grundlegende Konzept ist, haben führende Engines diesen Prozess weiterentwickelt, um die Codeoptimierung zu verbessern.

Eine solche Weiterentwicklung ist der Einsatz des Just-In-Time (JIT)-Compilers. Die JIT-Kompilierung zielt darauf ab, die Vorteile der Interpretation mit denen der Vorkompilierung zu verbinden. Während das Programm ausgeführt wird, identifiziert der JIT-Compiler die am häufigsten genutzten Code-Teile und wandelt diese in Maschinencode um. Dies kann entweder auf der Basis ganzer Methoden oder kleinerer Code-Segmente erfolgen. Anders als bei traditionellen Ansätzen wird nicht der gesamte Code vorab in Maschinencode übersetzt. Stattdessen wird zunächst nur der notwendige Teil des Codes kompiliert. Wird eine Funktion aufgerufen, die noch nicht in Maschinencode vorliegt, erfolgt die Umwandlung just in dem Moment. Diese Methode reduziert die Belastung der CPU erheblich (freeCodeCamp, 2020).

Ein entscheidender Vorteil des JIT-Compilers ist seine Fähigkeit, Maschinencode zur Laufzeit zu generieren, der speziell für die CPU-Architektur des ausführenden Computers optimiert ist. Diese Optimierung führt zu einer signifikanten Steigerung der Ausführungsgeschwindigkeit und Effizienz von JavaScript-basierten Anwendungen (CRYREX Enterprise, 2023).

#### Call Stack & Execution Context

Obwohl wir JavaScript häufig als asynchrone Sprache beschreiben, trifft dies nicht auf die Engine zu. Im Kern ist die JavaScript-Engine nämlich Single-Threaded, was bedeutet, dass Befehle sequenziell abgearbeitet werden (Kamal, 2023). Ein zentrales Element dabei ist der Call Stack, der die Execution Contexts verwaltet und diese nacheinander abarbeitet. Dabei folgt er einem Last-In-First-Out-Konzept und verarbeitet immer den Execution Context und seine Anweisungen, der zuletzt auf ihm abgelegt wurde. Der globale Execution Context bildet die Grundlage, in der der JavaScript-Code startet, und befindet sich somit an der Basis des Call Stacks. Diese Umgebung beinhaltet alle globalen Variablen und Funktionen, die in einem Skript definiert sind. Funktionale Execution Contexts entstehen jedes Mal, wenn eine Funktion aufgerufen wird. Jeder dieser Kontexte verfügt über seine eigene Umgebung zur Definition und Verwaltung von Variablen und Funktionen. Ein Execution Context durchläuft zwei Phasen: die Erstellungsphase und die Ausführungsphase. In der Erstellungsphase wird Speicherplatz für Variablen und Funktionen reserviert. Funktionen werden dabei vollständig gespeichert, während Variablen zunächst als 'undefined' initialisiert werden. Während der Ausführungsphase wird der Code Zeile für Zeile abgearbeitet. Bei jedem Funktionsaufruf wird ein neuer funktionaler Execution Context erstellt, der seinen eigenen Platz im Call Stack erhält. Beim Start eines JavaScript-Programms wird zuerst der globale Execution Context auf den Call-Stack gelegt. Jeder Funktionsaufruf führt zur Erstellung eines neuen funktionalen Execution Contexts durch die JavaScript-Engine, der auf den Stack gelegt wird. Die Engine führt die Funktion innerhalb dieses Kontextes aus. Nach Abschluss der Funktion wird ihr Execution Context vom Stack genommen, und die Ausführung setzt sich im vorherigen Kontext fort. Der Call-Stack folgt dabei dem Prinzip Last In, First Out, was bedeutet, dass die zuletzt hinzugefügte Funktion als erste wieder entfernt wird (Bera, 2022) (Kolosovskyi, 2016).

#### Der Event Loop

Trotz dieser grundsätzlich synchronen Natur besitzt JavaScript asynchrone Fähigkeiten durch Web-APIs, die es ermöglichen, Operationen im Hintergrund durchzuführen, während der Hauptthread aktiv bleibt (Kamani, 2016). Dies ist besonders wichtig, um zu verhindern, dass rechenintensive Aufgaben oder Funktionen, die auf ein Ergebnis warten, den Call Stack blockieren. Die Web-API - die nicht direkt Teil von JavaScript, sondern eine Erweiterung darüber hinaus ist - wird vom Browser bereitgestellt und kann über JavaScript angesprochene Schnittstellen genutzt werden. Zu den Funktionen der Web-API gehören beispielsweise die Manipulation des DOM, die Abfrage der Geolokation eines Geräts, Web-Socket-Kommunikation, Web-Anfragen und die Nutzung von Web-Workern für das Multithreading in JavaScript (Sagore, 2019).

Um die asynchronen Fähigkeiten von JavaScript durch die Web-API zu verstehen, ist der Event-Loop als weiteres Grundkonzept der JavaScript-Engine essenziell. Der Event-Loop funktioniert nach einem einfachen Prinzip: In einer endlosen Schleife wartet die JavaScript-Engine auf Aufgaben, führt diese aus und wartet dann in einem Ruhezustand auf weitere Aufgaben. Anhand von Abbildung 2 wird das Zusammenspiel zwischen der JavaScript-Engine, der Web-API und dem Event-Loop verdeutlicht. Eine Funktion auf dem Call Stack verwendet die setTimeout-Funktion der Web-API und wird anschließend vom Call Stack entfernt, ohne diesen weiter zu blockieren. Die Web-API initiiert einen eigenen Thread, der die angegebene Zeit abwartet, um danach in der Task Queue einen Callback zu registrieren. Dieser Callback enthält den Code, der nach dem Ablauf des setTimeout ausgeführt werden soll. Der Event-Loop überprüft in regelmäßigen Abständen, ob der Call Stack leer ist; sofern dies zutrifft, wird der Code des Callbacks auf den Call Stack gelegt und ausgeführt. Dieses Konzept ermöglicht es, asynchrone Funktionen wie Web-Anfragen sowie Event-Handler und Berechnungen in eigenen Threads effizient zu verarbeiten (Verma, 2021).

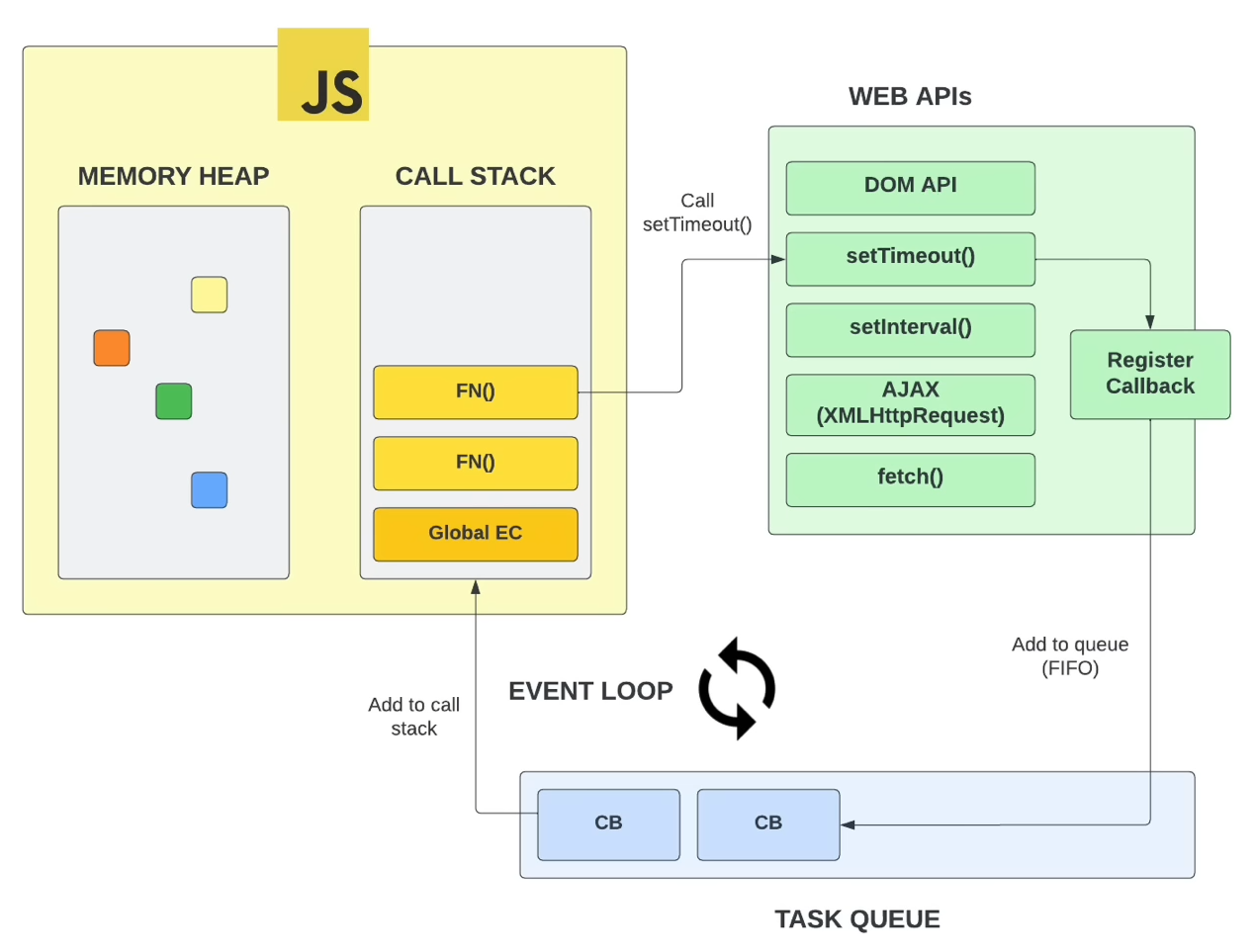


Abbildung 2: Zusammenspiel von Event Loop, Web-API und JS-Engine (Traversy Media, 2022)

#### Memory Storage

Im Vergleich zu C++, einer kompilierten Programmiersprache, die von Entwicklern eine manuelle Verwaltung des Speichers verlangt, bietet JavaScript eine automatische Speicherverwaltung. Dieser Prozess - bekannt als Garbage Collection - enthebt den Entwickler der Aufgabe, Speicher explizit zu allokieren und freizugeben (mdn web docs, kein Datum). JavaScript unterscheidet zwischen primitiven Datentypen und Referenzdatentypen. Primitive Typen, wie Strings, Zahlen oder Booleans und Referenztypen wie Objekte, Arrays und Funktionen. Primitive Typen werden direkt im Stack gespeichert, einem Speicherbereich mit fester Größe, der sich durch eine LIFO (Last In, First Out)-Struktur auszeichnet. Diese Struktur ermöglicht einen schnellen und effizienten Zugriff auf die gespeicherten Werte. Referenztypen hingegen werden im Heap abgelegt, einem dynamischen und größer dimensionierten Speicherbereich, der für komplexere Datentypen reserviert ist. Während die Größe und Struktur dieser Daten variieren kann, wird auf dem Stack lediglich eine Referenz auf die tatsächlichen Werte im Heap gespeichert (Onwuemene, kein Datum).

In Abbildung 3 wird das Konzept dargestellt: Die Variable 'name', die den primitiven Datentyp String besitzt, wird direkt im Stack abgelegt. Demgegenüber finden Funktionen wie 'getOwner' sowie Objekte wie 'newPerson', 'dog' und 'person' ihren Speicherplatz im Heap. Auf dem Stack wird für diese lediglich eine Referenz hinterlegt, die auf ihre Position im Heap zeigt. Ein wichtiger Aspekt hierbei ist, dass bei der Erstellung eines neuen Objekts und der Zuweisung eines anderen Objekts zu diesem nicht das vollständige Objekt dupliziert wird, sondern ausschließlich die Referenz. Diese Vorgehensweise wird in Abbildung X verdeutlicht, indem die Variablen 'person' und 'newPerson' identisch auf dasselbe Objekt im Heap verweisen.

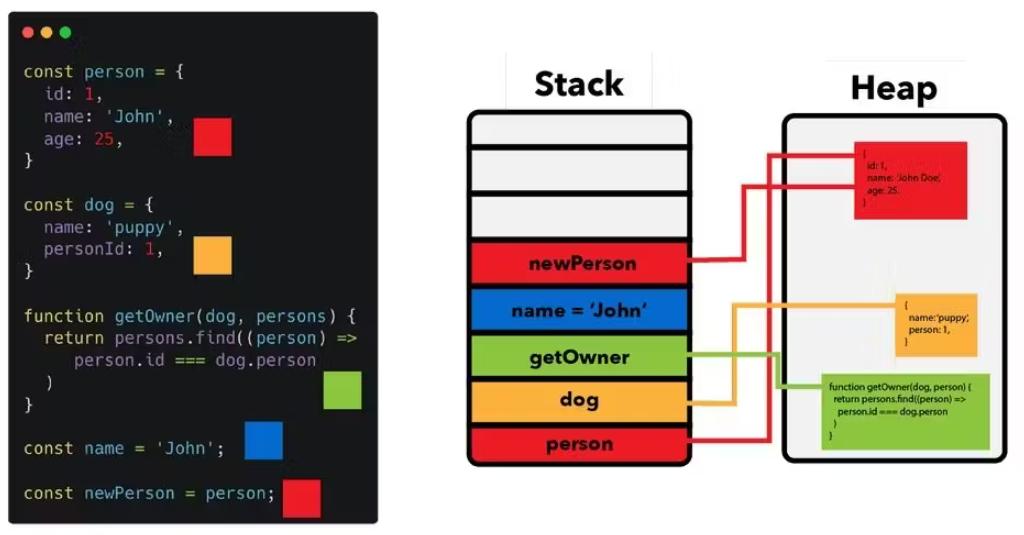


Abbildung 3: Speichermanagement in JavaScript (Onwuemene, kein Datum)

## WebAssembly – ein optimiertes Format für die JavaScript Engine

WebAssembly (Wasm) stellt ein weiteres Bytecode-Format dar, das besonders für Stack-basierte Engines, wie jene von JavaScript, optimiert wurde. Bytecode beschreibt Anweisungen, die zur Ausführung durch Software, wie beispielsweise eine virtuelle Maschine, entworfen sind und nicht direkt von der Hardware verarbeitet werden. Dieser Code ist kompakter als Quellcode, allerdings in der Regel für Menschen nicht direkt lesbar. Bytecode fungiert als Zwischenschritt zwischen dem ursprünglichen Quellcode und dem Maschinencode, welcher von der Hardware ausgeführt werden kann (Gallant, Meet WebAssembly, 2019). Ein Beispiel einer anderen Programmiersprache, die zu Bytecode kompiliert wird, ist Java (java T point, kein Datum). Eine Besonderheit von Wasm ist, dass Entwickler normalerweise nicht direkt in WebAssembly programmieren, sondern in einer höheren Programmiersprache, die üblicherweise in Maschinencode kompiliert wird. Eine solche Sprache ist beispielsweise C++. Um Wasm-Code mittels C++ zu erzeugen, wird zusätzliche Software benötigt, die für das Transpilieren des Codes verantwortlich ist. Im Falle von C++ übernimmt die Software *Emscripten* diese Aufgabe (emscripten, 2015).

Wie in Abbildung 4 dargestellt wird JavaScript-Quellcode direkt an den Browser übermittelt, der ihn interpretiert und in Maschinencode umwandelt. Im Unterschied dazu wird Wasm-Code hingegen bereits im Vorfeld vom Entwickler von C++ Quellcode in Wasm transpiliert und optimiert, sodass der Browser nur noch mit dem Wasm Bytecode arbeitet, diesen interpretiert und in Maschinencode umsetzt (mdn web docs, kein Datum) (Gallant, Meet WebAssembly, 2019).

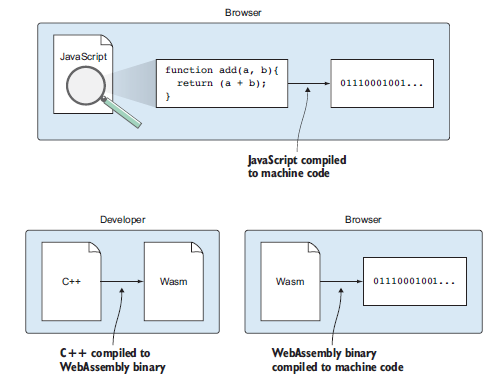


Abbildung 4: Unterschiede in der Kompilierung zwischen JavaScript im Vergleich zu WebAssembly (Gallant, Meet WebAssembly, 2019)

Wasm kann von verschiedenen JavaScript-Engines durch den JIT-Compiler interpretiert werden (WebAssembly, kein Datum). Im Unterschied zu JavaScript, wo die Variablentypen zur Laufzeit bestimmt und daher vom Browser dynamisch überwacht werden müssen, sind die Typen in Wasm bereits im Vorfeld festgelegt. Diese vorgegebene Typisierung reduziert die Notwendigkeit einer laufenden Codeüberwachung durch den Browser. Dadurch können Überprüfungen, die normalerweise zur Laufzeit erfolgen, entfallen, was zu einer schnelleren Ausführung führt. Zudem gewährleistet die Transpilierung eine Voroptimierung des Codes. Die damit einhergehenden Vorteile werden in Abbildung 5 illustriert. Sie zeigen auf, wie das Wasm-Modul – im Gegensatz zu JavaScript-Code – unmittelbar an den JIT-Compiler weitergeleitet werden kann, wodurch der Parser und Interpreter der JavaScript-Engine umgangen werden. Dies ermöglicht es WASM, direkt in Maschinencode kompiliert zu werden, was wiederum einen deutlichen Leistungsvorteil gegenüber herkömmlichem JavaScript-Code bietet. Außerdem ist Bytecode deutlich kompakter als Quellcode, was die Übertragung beschleunigt. Die JavaScript-Engine muss sich zudem nicht um die Garbage Collection kümmern, da dies im Wasm-Code manuell gehandhabt wird. Dadurch kann Wasm im Vergleich zu JavaScript einen signifikanten Leistungsvorteil bieten und fast so schnell wie nativer Code laufen (Clark, 2017).

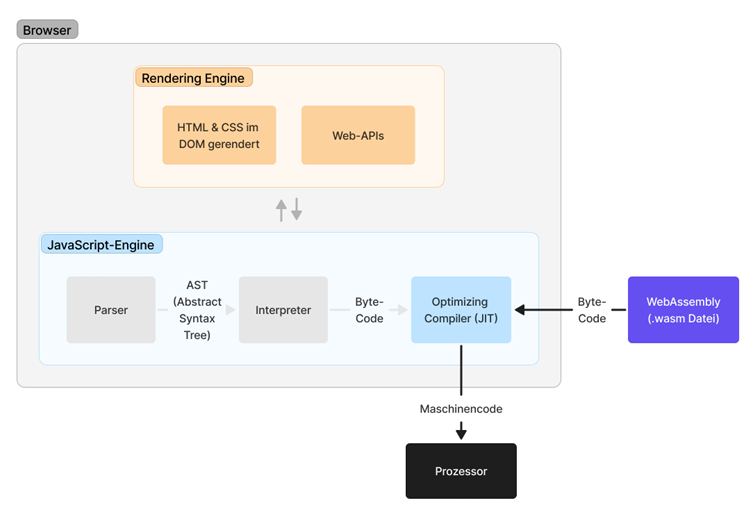


Abbildung 5: WebAssembly im Browser, schematische Darstellung (Schnarr, 2023)

Derzeit nutzen Entwickler WebAssembly für zwei Hauptzwecke: um ihre bestehende Codebasis ins Web zu portieren, wie zum Beispiel Figma, Google Earth oder AutoCAD, und um rechenintensiven Teilprozessen in JavaScript einen Leistungsschub zu verleihen (Chromium Blog, 2019) (Wallace, 2017) (WebAssembly, 2019).

Wasm wird stets in Modulen verwendet. Ein Wasm-Modul wird durch eine Wasm-Datei repräsentiert und kann durch eine Browser-API in JavaScript initialisiert werden. Bei dieser Initialisierung kann ein ArrayBuffer übergeben werden, der im Modul als linearer Speicher dient. Dies ermöglicht das Kernkonzept der Sicherheit einer Sandbox-Umgebung: Wasm-Module haben keinen direkten Zugriff auf den Speicher des Geräts, sondern nur auf den ihnen zugewiesenen ArrayBuffer. Mehrere Instanzen von Wasm-Modulen können sogar denselben linearen Speicher über SharedArrayBuffers teilen, um Module dynamisch miteinander zu verknüpfen. Um mit JavaScript zu interagieren, importiert und exportiert das Modul bei der Initialisierung bestimmte Funktionen, sodass Funktionen im Modul aus JavaScript aufgerufen werden können und umgekehrt. Da ein Wasm-Modul als Sandbox-Umgebung innerhalb von JavaScript läuft, kann es nicht direkt auf die Web-API zugreifen und somit das DOM nicht direkt manipulieren. Ein Wasm-Modul benötigt daher immer JavaScript-Code als Vermittler, um mit dem DOM oder anderen Web-API's zu interagieren. Die Interaktion mit einer JavaScript-Engine, wie z.B. V8, wird oft so beschrieben, dass in der Host-Runtime ein Gast-Wasm-Modul eingebunden wird. Die Host-Runtime fungiert als das vertrauenswürdige Umfeld (Trusted Environment), wobei das Wasm-Gastmodul nur limitierte Privilegien in diesem Umfeld besitzt (mdn web docs, kein Datum) (Nadiger, 2023).

## Multithreading

Multithreading ermöglicht es, verschiedene Threads zeitgleich in einem Prozess zu bearbeiten. Ein Prozess ist Code beziehungsweise ein Programm, das in einen isolierten Speicherbereich geladen wird und ohne Zustimmung des Kernels keinen anderen Speicherbereich adressieren kann. Das Starten eines Progamms, wie beispielsweise eines Browsers, initiiert einen solchen Prozess. Ohne zusätzliche Threads oder Prozesse wird immer nur eine Anweisung nach der anderen ausgeführt, in der vom Programmcode vorgesehenen Reihenfolge. Eine Anweisung entspricht in der Regel einer Zeile im Maschinencode des Prozessors. Ein Programm kann zusätzliche Prozesse erzeugen, die jeweils über ihren eigenen Speicherbereich verfügen. Diese Prozesse teilen sich keinen Speicher und haben eigene Befehlszeiger, was bedeutet, dass jeder von ihnen gleichzeitig eine andere Anweisung ausführen kann. Ein Prozess kann aber auch mehrere Threads erzeugen. Ein Thread ist eine Sequenz von programmierten Anweisungen, die unabhängig von einem Scheduler verwaltet werden kann (Kavi, 1998) (Hunter & English, What Are Threads?, 2021).

Wie in Abbildung 6 dargestellt, ermöglicht Multithreading die Zuordnung von CPU-Ressourcen zu mehreren Threads, wodurch deren gleichzeitige Verarbeitung ermöglicht wird. Abbildung 7 demonstriert, dass die Threads innerhalb eines Prozesses existieren und sich den Speicher teilen, jedoch über eigene Call-Stacks verfügen, inklusive separater Stack Register und Programm Counter. Durch Multithreading wird somit die parallele Ausführung mehrerer Threads in einem einzigen Prozess realisiert (Sonal, 2020).

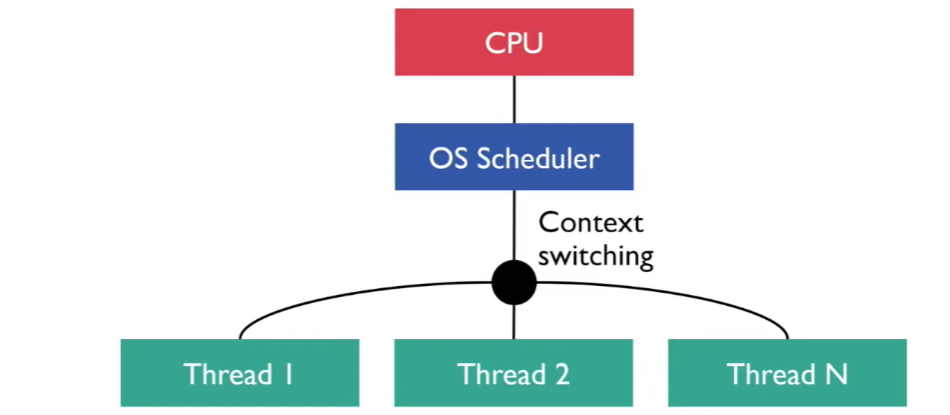


Abbildung 6: Multithreading und gleichzeitige Verarbeitung mehrerer Threads (Mihaylov, 2019)

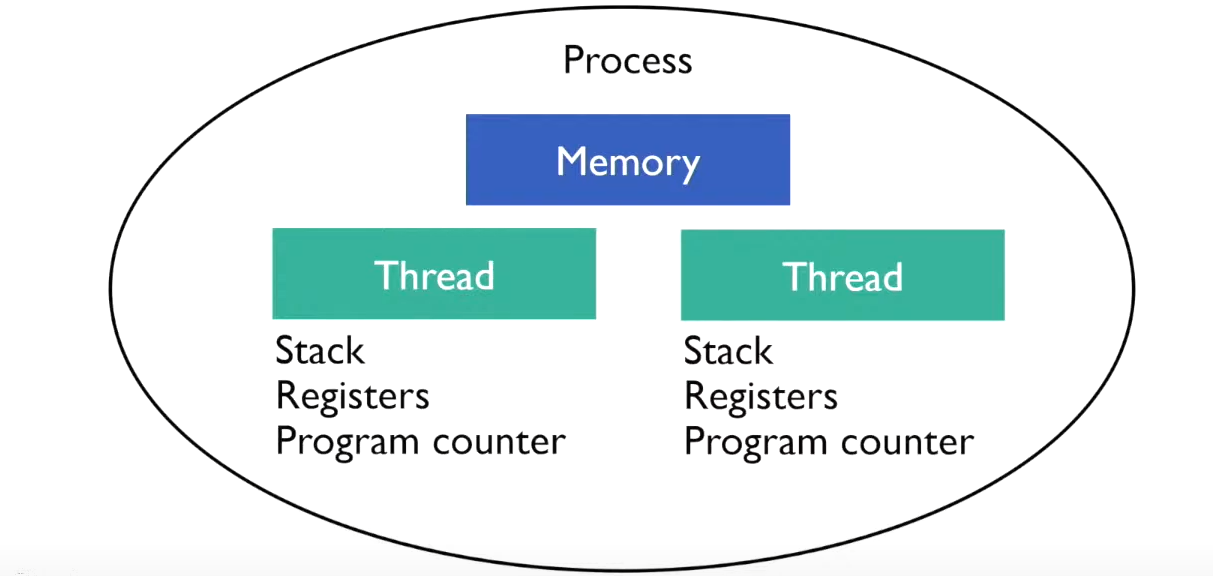


Abbildung 7: Multithreading in einem Prozess mit geteiltem Speicher und individuellen Call-Stacks (Mihaylov, 2019)

### Speichernutzung in Threads

Die gemeinsame Speichernutzung durch Threads führt zu Herausforderungen in der Speicherverwaltung, da dadurch simultane Zugriffe auf denselben Speicherbereich möglich sind. In Abbildung 8 wird diese Problematik durch einen Entwickler-Post wunderbar verdeutlicht.

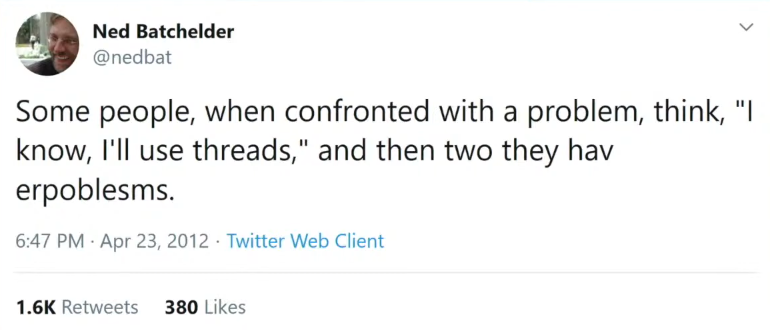


Abbildung 8: Entwicklerbeitrag zur Problematik der gemeinsamen Speichernutzung durch Threads (Batchelder, 2012)

Um dieses als Race Condition bekannte Problem zu lösen, ist eine Synchronisierung des Speicherzugriffs zwischen den Threads erforderlich. Dies gelingt, indem man ausgewählte Speicherbereiche jeweils nur einem Thread zugänglich macht. Durch den Einsatz von atomaren Operationen ist es möglich, spezifische Speicherbereiche vorübergehend zu blockieren, bis eine Abfolge von Operationen vollständig durchgeführt wurde. Dadurch wird verhindert, dass andere Threads in der Zwischenzeit auf diese Bereiche zugreifen können, bis sie wieder freigegeben werden (OpenCSF, kein Datum).

### Multithreading in JavaScript und Webassembly

Wie bereits dargestellt wurde, basiert JavaScript im Kern auf einem einzigen Thread. Das bedeutet, dass rechenintensive Codeabschnitte diesen Thread blockieren können, was zu dem Problem führen kann, dass Webseiten einfrieren. Sie reagieren nicht mehr auf Eingaben, da die Callbacks nicht in den Call-Stack übernommen werden können, weil dieser bereits ausgelastet ist.

Eine Lösung für dieses Problem bieten so genannte Web Worker, die von der Browser-API bereitgestellt werden. Diese ermöglichen es, JavaScript-Code vollständig unabhängig in einem separaten Thread auszuführen. Jeder dieser Threads verfügt über einen eigenen Call-Stack und eigenen Speicher, wodurch er fast vollständig isoliert vom Rest der Anwendung agiert. Aus dieser gezielten Isolation ergibt sich allerdings die Einschränkung, dass Web Worker keinen Zugang zum DOM-Objekt haben, da dieses ausschließlich im globalen Kontext zugänglich ist (mdn web docs, kein Datum).

Abbildung 9 illustriert die Kommunikationsweise zwischen verschiedenen Threads in JavaScript. Die Funktion postMessage erlaubt den Austausch von Nachrichten zwischen Threads. Dabei werden Objekte, die zwischen Threads gesendet werden, geklont und nicht geteilt. Das bedeutet, dass bei der Übermittlung eines Objekts an einen Thread zwei getrennte Instanzen dieses Objekts existieren. Dies kann besonders ineffizient sein, wenn man große Objekte zwischen Threads austauschen möchte. Außerdem entspricht dies nicht vollständig dem Multithreading-Paradigma, da die Threads nicht auf denselben Speicher zugreifen, sondern jeder Thread über seinen eigenen, isolierten Speicher verfügt (Kitamura & Ubl, 2012).

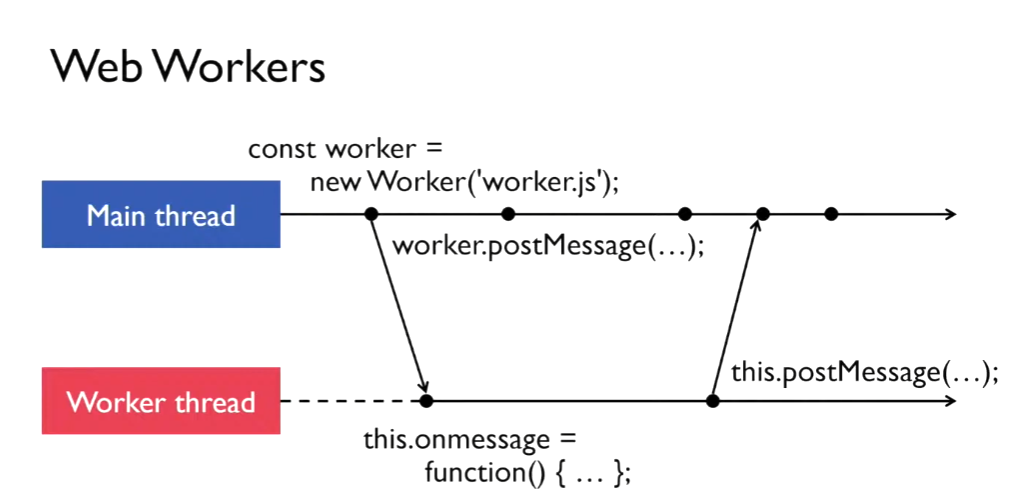


Abbildung 9: Thread-Kommunikation in JavaScript durch postMessage, wobei Objekte geklont statt geteilt werden (Mihaylov, 2019)

In Programmiersprachen wie C++ oder Rust stehen uns deutlich fortgeschrittenere Möglichkeiten des Multithreadings zur Verfügung, etwa durch die Nutzung der C++ -API P-Threads (POSIX Threads). Sie ermöglicht es, Threads auf einer hardwarenahen Ebene zu steuern, da sie direkt mit dem Betriebssystem interagiert, um Multithreading-Fähigkeiten bereitzustellen (Ippolito, 2004). Wenn wir jedoch C++-Code zu WebAssembly transpilieren und diesen dann über den JIT-Kompiler der JavaScript-Engine ausführen möchten, fehlt uns der Zugang zu den System-Schnittstellen, die druch P-Threads verwendet werden. Um dennoch mit der Bibliothek arbeiten zu können, bietet Emscripten eine API-kompatible Implementierung von P-Threads an, die auf Web Workern, Shared Memory und Atomic Operations basiert. So kann derselbe Code im Web funktionieren, ohne dass Änderungen notwendig sind. Dabei wird sogenannter JavaScript-„Gluecode“ durch Emscripten generiert, der dafür sorgt, das WebAssembly-Modul zu initialisieren und mit den notwendigen Ressourcen auszustatten. Das bedeutet, dass der Einsatz von Multithreading in WebAssembly, Stand 2024, intern immer durch die Web Worker der Browser-API realisiert wird (emscripten, kein Datum).

### Javascripts Konkurrenzmodelle

Durch die vorgestellten Ansätze zur gemeinsamen Benutzung des Arbeitsspeichers duch die Threads in JavaScript ergeben sich zwei Architekturmodelle für die Ausführung außerhalb des Hauptthreads – beide nutzen Web Worker, allerdings auf unterschiedliche Art und Weise.

**Das Aktorenmodell:**

Im Aktorenmodell verfügt jeder Web Worker über seinen eigenen Speicher, auf dem er operiert. Kein anderer Thread hat Zugriff darauf, was die Notwendigkeit für Speichersynchronisation durch atomare Operationen überflüssig macht. Die einzelnen Aktoren können lediglich Nachrichten austauschen und auf diese reagieren. Bei den Web Workern wird dieser Nachrichtenaustausch durch die bereits diskutierte postMessage-Funktionalität realisiert. Hierbei kann jedoch der Nachrichtenaustausch zum Engpass der Anwendung werden, da jede Nachricht durch den "structured clone"-Algorithmus geklont werden muss (Surma, The State Of Web Workers In 2021, 2021). Benchmark-Tests haben gezeigt, dass die Größe der Nachricht proportional zur Übertragungszeit ist und selbst auf leistungsschwachen Geräten bei Nachrichtengrößen von 102kB unter 100ms pro Transfer bleibt (Surma, Is postMessage slow?, 2019).

**Das „shared memory“ Modell:**

Wie bereits beschrieben, basiert der traditionelle Multithreading-Ansatz auf gemeinsam genutztem Speicher. Dieser Ansatz war in JavaScript lange Zeit nicht praktikabel, da fast alle Browser-APIs unter der Annahme entwickelt wurden, dass kein gleichzeitiger Zugriff auf Objekte erfolgt. Dieses Feature wurde jedoch immer wichtiger, insbesondere für Technologien wie WebAssembly, die auf dem Paradigma des „shared memory“ zwischen Modulen basieren, da dieses Paradigma auch in C++ und Rust nutzbar ist. Um diese Technologien zu unterstützen, wurde der gemeinsam genutzte Speicher für einen speziellen Typen ermöglicht: das SharedArrayBuffer. Dieser Buffer kann an Web Worker übergeben werden und somit von jedem Thread verändert werden. Diese Änderungen sind dann in jedem anderen Thread verfügbar, was uns jedoch dazu zwingt, die daraus resultierenden Race-Condition-Probleme zu lösen (Surma, The State Of Web Workers In 2021, 2021). Basierend auf diesem Architekturansatz realisieren Transpilierungswerkzeuge wie Emscripten das Multihreading innerhalb von WebAssembly. Die Wasm-Module werden mit dem SharedArrayBuffer initialisiert. Anschließend wird für jeden P-Thread innerhalb des Wasm-Moduls ein eigener WebWorker gestartet. Jeder dieser WebWorker wird ebenfalls mit dem SharedArrayBuffer initialisiert und erhält die spezifischen Teile des Wasm-Moduls, die im jeweiligen P-Thread ausgeführt werden sollen (emscripten, kein Datum).

### Thread Pooling

Die Verwendung von Web Workern zur Erstellung von Threads ist ein Prozess, der viele Ressourcen beansprucht und bei einer übermäßigen Anzahl generierter Threads ineffizient wird. Dies lässt sich darauf zurückführen, dass ab einem bestimmten Grad der CPU-Auslastung der Leistungszuwachs stagniert und die Generierung zusätzlicher Threads eher Ressourcen beansprucht, als dass sie effektive Vorteile durch Parallelverarbeitung bietet (Wilson, 2018). Um diesem Problem zu begegnen, wird empfohlen, nicht für jede einzelne Aufgabe einen neuen Thread zu erstellen, sondern das Prinzip der Wiederverwendung von Threads durch sogenannte Thread Pools zu nutzen.

Innerhalb eines Thread Pools wird eine bestimmte Anzahl von Threads vorgehalten, die, nachdem sie eine Aufgabe abgeschlossen haben, nicht beendet, sondern für die Bearbeitung neuer Aufgaben wiederverwendet werden. Diese Strategie führt zu einer verbesserten Gesamtleistung, da sie den Overhead, der mit der Erstellung und Zerstörung von Threads verbunden ist, erheblich reduziert. Thread Pools bringen bedeutende Vorteile mit sich, darunter eine effiziente Verwaltung der Ressourcen, eine gesteigerte Leistungsfähigkeit und eine erhöhte Skalierbarkeit. Allerdings stellen sie auch Herausforderungen dar, wie eine Zunahme der Komplexität des Codes und Schwierigkeiten beim Management von Abhängigkeiten zwischen verschiedenen Aufgaben (geeks for geeks, 2024).

# Stand der Technik

In diesem Kapitel widmen wir uns der Analyse der aktuellen Entwicklungen und Anwendungsbereiche von WebAssembly und Multithreading, insbesondere im industriellen Kontext. Dabei werden spezifische Einsatzgebiete beleuchtet, in denen WebAssembly und Multithreading bereits heute einen signifikanten Einfluss auf die industrielle Praxis haben. Außerdem analysieren wir bisherige Forschungsarbeiten, um Fortschritte als auch offene Forschungsfragen und Herausforderungen zu identifizieren. Dies dient nicht nur der Einordnung dieser Arbeit in den aktuellen Forschungsstand, sondern auch der Identifikation von Potenzialen für innovative Beiträge im Bereich WebAssembly in Verbindung mit Multithreading.

## Anwendungsbeispiele von WebAssembly in der Industrie

WebAssembly wird in einer Vielzahl von Projekten eingesetzt, um bestehende Software für das Web zu optimieren oder die Leistung von Webanwendungen durch Umstellung ressourcenintensiven Codes auf WebAssembly zu steigern. WebAssembly findet Anwendung in zahlreichen renommierten Projekten, darunter (Webassembly, kein Datum):

* **Unity**: Unity ist eine plattformübergreifende Spiele-Engine, die über eine Web-Exportfunktion verfügt, welche durch WebAssembly angetrieben wird. Dies ermöglicht die Erstellung von hochleistungsfähigen Spielen direkt im Webbrowser ohne den Bedarf an zusätzlichen Plugins.
* **Figma**: Figma - ein kollaboratives Interface-Design-Tool - nutzt WebAssembly, um eine nahtlose und leistungsstarke Benutzererfahrung zu bieten, indem es Design-Prozesse direkt im Browser ermöglicht.
* **Blazor**: Mit Blazor können Entwickler Client-Webanwendungen mit C# erstellen, angetrieben durch WebAssembly.
* **OpenCV**: Als eine beliebte Bibliothek für Computer Vision und Bildanalyse, ermöglicht OpenCV durch WebAssembly die Implementierung komplexer Bildverarbeitungsaufgaben direkt im Webbrowser.
* **QT**: QT, ein freies und Open-Source-Widget-Toolkit zur Erstellung grafischer Benutzeroberflächen sowie plattformübergreifender Anwendungen, profitiert ebenfalls von WebAssembly, um leistungsfähige Anwendungen im Web bereitzustellen.
* **Kubewarden**: Ein Kubernetes Admission Controller, der WebAssembly nutzt, um Anfragen zu validieren. Dies zeigt die Vielseitigkeit von WebAssembly, die über traditionelle Webanwendungen hinausgeht und in die Verwaltung und Sicherheit von Cloud-Infrastrukturen einfließt.
* **Google** **Earth**: Google Earth bietet eine 3D-Darstellung der Erde, die hauptsächlich auf Satellitenbildern basiert und durch WebAssembly im Web gerendert wird, was eine beeindruckende Visualisierung der Erde direkt im Browser ermöglicht.

In den nächsten Abschnitten gehen wir auf die konkrete Implementierung von drei weiteren Webanwendungen ein, die WebAssembly in Verbindung mit Multithreading nutzen. Außerdem wird, falls möglich, die gewählte Architektur für das Zusammenspiel der Technologien beleuchtet.

### Ebay

Ebay ist eine der größten E-Commerce-Webseiten der Welt, und gerade in diesem Bereich ist das Aufkommen der Technologie WebAssembly überraschend. Denn die Technologien, die im E-Commerce-Bereich eingesetzt werden, sind recht simpel und benötigen normalerweise nicht viel Rechenleistung. Doch Ebay besaß native Anwendungen für iOS und Android, die den Web-Nutzern gegenüber einen Vorteil boten: einen Barcode-Scanner. Diese Funktion nutzt die Kamera des Geräts, um den Barcode eines Produkts zu scannen und die Produktdaten automatisch zu erfassen. Dieser Barcode-Scanner erforderte eine intensive Bildverarbeitung auf dem Gerät, um die Barcode-Nummer aus dem Kamera-Stream zu erkennen. Auf den nativen Apps war dieses Feature durch die Verwendung optimierter C++-Bibliotheken gut nutzbar. Leider war das Feature für Web-Nutzer nicht verfügbar, da die Bildverarbeitung in JavaScript für die meisten Geräte zu rechenintensiv war, sodass nur etwa 20 % der Nutzer es im Web performant nutzen konnten.

An diesem Punkt kam WebAssembly ins Spiel, um eine leistungsstarke Option für das Web zu bieten. Zunächst versuchten die Entwickler, ihre bereits funktionierende native Lösung nach WebAssembly zu transpilieren und diese aus JavaScript heraus aufzurufen. Diese Umsetzung war sehr leistungsfähig und erreichte sogar 50 FPS im Vergleich zur nativen Variante, die nur mit 1 FPS lief. Jedoch erzielte sie nicht genauso gute Ergebnisse, da der Barcode nur in 60 % der Fälle korrekt gelesen wurde. Die Entwickler erkannten, dass die native Variante besser auf die Kamerafunktionen abgestimmt war und den Barcode somit besser fokussieren konnte. Daher probierten sie eine andere native Bibliothek aus, die besonders gut mit unscharfen Bildern umgehen konnte, nämlich ZBar.

ZBar wurde zu WebAssembly transpiliert und in die Webanwendung integriert. Die Leistung war mit rund 15 FPS akzeptabel, und die Erfolgsquote lag nahe bei 80 %, was definitiv eine Verbesserung darstellte, aber immer noch nicht 100 % zuverlässig war. Beim Vergleich der beiden Varianten stellten die Entwickler fest, dass in den Fällen, in denen ZBar nicht rechtzeitig ein Ergebnis lieferte, ihre eigene C++-Bibliothek schnell ein richtiges Ergebnis liefern konnte.

Dies führte zu der Idee, alle drei Möglichkeiten - ZBar, die eigene Bibliothek und die JavaScript-Bibliothek - parallel zu nutzen und auf drei Threads laufen zu lassen. Wenn ein Thread ein Ergebnis liefert, werden alle anderen Threads beendet. In Abbildung 10 ist die gewählte Architektur für die Anwendung dargestellt. Der Hauptthread startet drei Web-Worker, die dann über Glue-Code entweder ihr WebAssembly-Modul einbinden oder die native JavaScript-Bibliothek ansprechen. Dies zeigt einen interessanten Anwendungsfall, bei dem Multithreading nicht auf der Basis von WebAssembly-Code erfolgt, sondern auf der JavaScript-Seite, um verschiedene WebAssembly-Module zu laden, die intern nicht miteinander kommunizieren müssen (Padmanabhan & Jha, 2019).

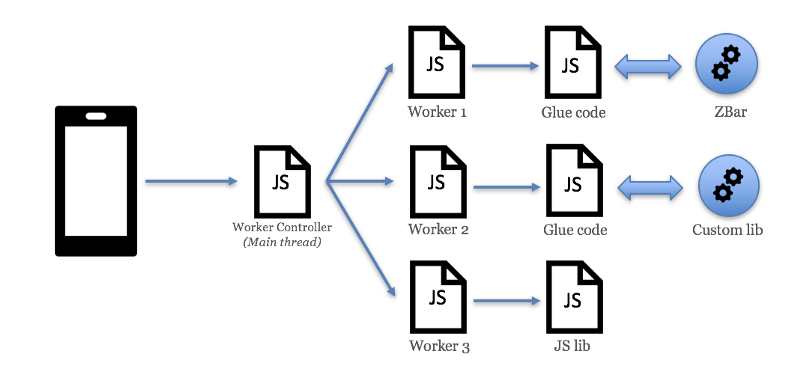


Abbildung 10: Anwendungsarchitektur mit Hauptthread und drei Web-Workern, die unabhängige Module laden (Padmanabhan & Jha, 2019)

### TensorFlow.js

TensorFlow.js ist eine von Google entwickelte JavaScript Bibliothek für maschinelles Lernen. Auf ihrem TensorFlow Blog erläutern die Entwickler, wie sie durch die Nutzung von WebAssembly, Multithreading und Single Instruction Multiple Data (SIMD) – einer parallelen Verarbeitungstechnik, bei der eine einzige Anweisung gleichzeitig auf mehrere Datenelemente angewendet wird – ihre Bibliothek um das Zehnfache leistungsfähiger machen konnten. Für maschinelles Lernen eignen sich besonders die Technologien Multithreading und WebAssembly, da es auf rechenintensive Operationen angewiesen ist, die hervorragend parallel verarbeitet werden können. Die Verbesserungen demonstrieren sie anhand ihrer KI BlazeFace, die Gesichtserkennung im Web ermöglicht (Yuan & Dukhan, 2020). Testen kann man die beiden Ansätze online unter: [optimierte Variante](https://tfjs-wasm-simd-demo.netlify.app/) und [unoptimierte Variante](https://storage.googleapis.com/tfjs-models/demos/blazeface/index.html?tfjsflags=WASM_HAS_MULTITHREAD_SUPPORT:false,WASM_HAS_SIMD_SUPPORT:false).

Abbildung 11 zeigt die Benchmark-Tests der Entwickler. Besonders interessant für diese Arbeit ist der Leistungsgewinn zwischen „Wasm + SIMD“ und „Wasm + SIMD + Threads“. Dies deutet darauf hin, dass die Anwendung von Multithreading-Techniken, in Kombination mit WebAssembly, eine signifikante Leistungssteigerung in praktischen Anwendungen ermöglicht.

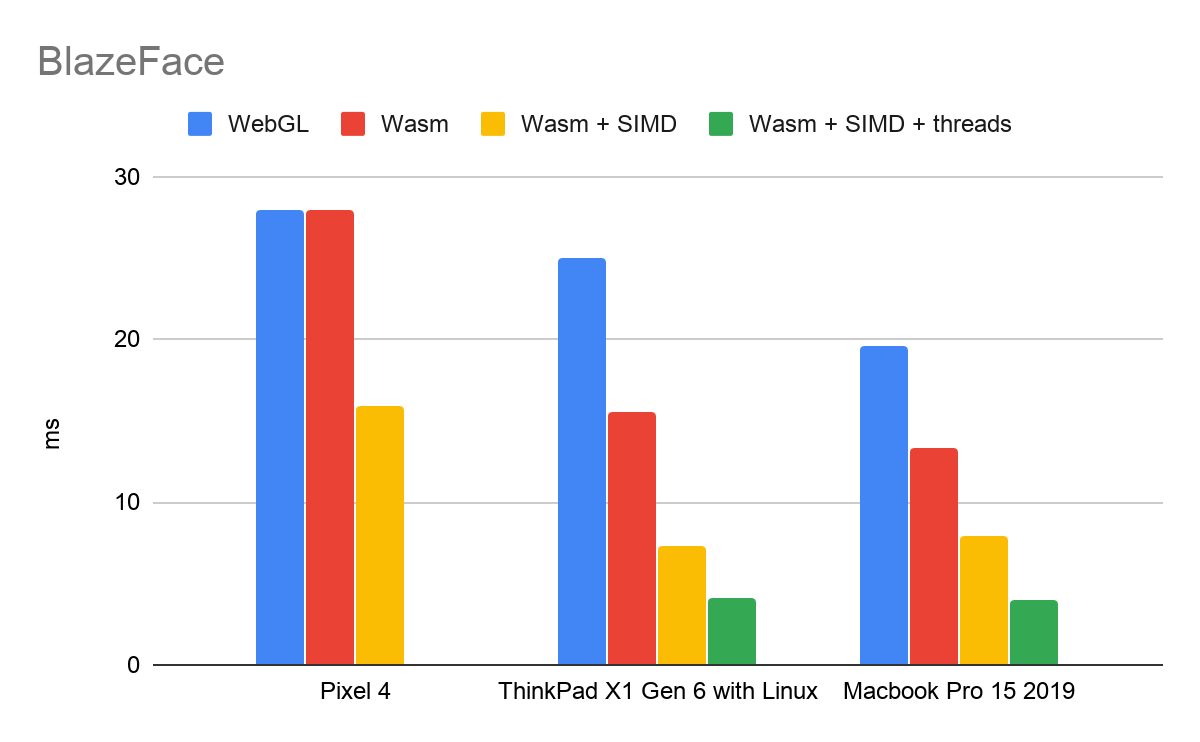


Abbildung 11: Benchmark-Tests mit Leistungsgewinn durch 'Wasm + SIMD + Threads' (Yuan & Dukhan, 2020)

In dem Entwickler Blog wird eine explizite Beschreibung zur Implementierung von Multithreading-Techniken weitgehend ausgespart. Jedoch lässt die Referenzierung von Wasm-Threads die Schlussfolgerung zu, dass innerhalb der WebAssembly-Module Multithreading zum Einsatz kommt. Dies impliziert, dass das Konzept des „Shared Memory“ zur Anwendung kommt.

### FFmpeg

FFmpeg ist ein umfassendes Multimedia-Framework, das nahezu alle Medienformate verarbeiten kann, einschließlich Dekodieren, Kodieren, Konvertieren, Mischen, Streamen, Filtern und Abspielen. Ursprünglich in C programmiert, wird FFmpeg von großen Unternehmen wie YouTube oder Netflix für die serverseitige Videobearbeitung eingesetzt (FFmpeg, kein Datum). Seit 2020 ist auch eine reine WebAssembly / JavaScript-Version verfügbar, die in zwei Varianten angeboten wird: eine mit Multithread- und eine mit Single-Thread-Verarbeitung (FFmpeg, kein Datum). In Abbildung 12 wird ersichtlich, dass das Single-Threaded Build „Core v0.12.3“ im Durchschnitt doppelt so lange für die Verarbeitung benötigt wie das Multithreading Build „Core-MT v0.12.3“. Interessanterweise zeigt die Abbildung auch, dass das native C-Build deutlich schneller ist als die Multithreaded WebAssembly-Version. Nach Angaben der Entwickler liegt dies daran, dass die WebAssembly-Version noch nicht vollständig optimiert wurde (FFmpeg, kein Datum).

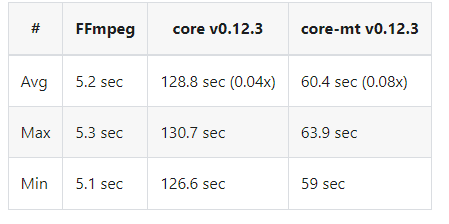


Abbildung 12: Vergleich von Verarbeitungszeiten zwischen Single-Threaded 'Core v0.12.3', Multithreading 'Core-MT v0.12.3' und einem schnelleren nativen C-Build (FFmpeg, kein Datum)

Die Implementierung des Multithreading-Features verdient besondere Aufmerksamkeit. Die Entwickler wählten einen differenzierten Ansatz gegenüber einer direkten Übertragung des existierenden Multithreading-Konzepts des C-Codes durch Emscripten in eine WebAssembly-äquivalente Lösung. Stattdessen wurden lediglich die zentralen Funktionen der C-Bibliothek nach WebAssembly transpiliert. In Abbildung 13 wird die spezielle Architektur für die Implementierung des Multithreadings dargestellt: Wenn wir die FFmpeg-Bibliothek im Hauptthread starten, interagieren wir ausschließlich mit einem FFmpeg-Worker-Thread über die 'postMessage'-Funktion für WebWorker. In diesem speziell entwickelten FFmpeg.worker wird dann der FFmpeg-Core geladen, der von C nach WebAssembly transpiliert wurde, um die Funktionalität im Browser bereitzustellen. In der Multithreading-Variante ist es möglich, aus dem FFmpeg-Worker heraus weitere FFmpeg-Core-WebWorker zu starten. Leider fehlen sowohl in der Grafik als auch im Repository Informationen darüber, wie diese verschiedenen FFmpeg-Core-Worker untereinander oder mit dem FFmpeg-Worker Nachrichten austauschen oder ihren Speicher teilen (FFmpeg, kein Datum).

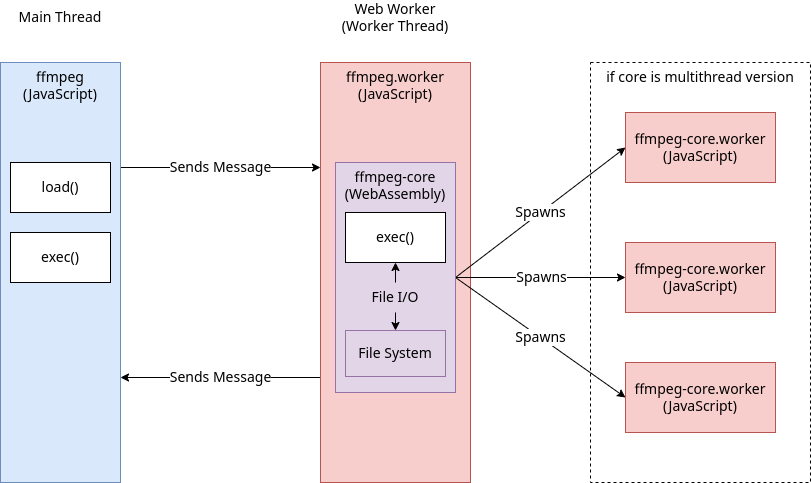


Abbildung 13: Architektur von FFmpeg-Multithreading mit Hauptthread und Worker-Kommunikation (FFmpeg, kein Datum)

## Verwandte Arbeiten

Dieser Abschnitt bietet einen umfassenden Überblick über relevante Forschungsarbeiten, die thematisch eng mit dieser Masterarbeit verbunden sind. Da die Performance von WebAssembly in dieser Arbeit eine zentrale Rolle spielt, analysieren wir Studien, die sich allgemeinen mit der Performance von WebAssembly und deren Besonderheiten befassen. Darüber hinaus werden aktuelle Implementierungsansätze untersucht, die das Zusammenspiel von WebAssembly und Multithreading ermöglichen. Abschließend wird ein Forschungspapier diskutiert, das die Konkurrenzmodelle des Aktorenmodells und des Shared-Memory-Modells gegenüberstellt und vergleicht. Durch diese Analysen sollen noch offene Forschungslücken identifiziert und die Notwendigkeit der Forschungsfrage dieser Arbeit unterstrichen werden.

### WebAssembly Performance

Um den Leistungsvorteil von WebAssembly vollständig zu erfassen und einzuordnen, ist es unerlässlich, die Arbeit "Understanding the Performance of WebAssembly Applications", veröffentlicht im November 2021 von Yan et al., eingehend zu analysieren (Yan, Tu, Zhao, Zhou, & Wang, 2021). Diese systematische Studie zielt darauf ab, ein tiefes Verständnis für die Performance von WebAssembly-Anwendungen zu entwickeln und diese mit der von JavaScript zu vergleichen. Im Verlauf ihrer Forschung haben Yan und sein Team vier wesentliche Erkenntnisse identifiziert, die in den folgenden Abschnitten erörtert werden.

#### WebAssembly-Compiler

Typischerweise werden WebAssembly-Programme generiert, indem Source-Code, geschrieben in Hochsprachen wie C/C++ und Rust, durch einen WebAssembly-Compiler wie Cheerp, Emscripten oder wasm-pack transpiliert wird. Diese WebAssembly-Compiler werden auf der Basis von LLVM[[1]](#footnote-1) entwickelt, deren Optimierungen jedoch nicht speziell auf WebAssembly zugeschnitten sind. Die Untersuchungen zeigen, dass solche Optimierungen oft ineffektiv für WebAssembly sind, was zu kontraintuitiven Ergebnissen führt. Da LLVM eine breite Palette von Optimierungen für verschiedene Zielplattformen bietet, die ursprünglich für die Kompilierung zu nativem Code konzipiert waren, können diese Optimierungen nicht immer die spezifischen Anforderungen und Eigenschaften von WebAssembly vollständig berücksichtigen. Die Herausforderung besteht darin, dass viele der LLVM-Optimierungen auf Annahmen basieren, die in einem Web-Kontext nicht unbedingt gültig sind. Im Vergleich der beiden WebAssembly-Compiler stellen die Autoren heraus, dass Emscripten gegenüber Cheerp im Allgemeinen performanter ist, jedoch mehr Speicher benötigt.

#### JIT-Optimierung

Die Autoren stellen heraus, dass die JIT-Optimierung einen signifikanten Einfluss auf die Leistung von JavaScript hat. Allerdings wurde keine wesentliche Leistungssteigerung für WebAssembly mit JIT beobachtet.

#### Speicherverbrauch und Performance

WebAssembly verwendet deutlich mehr Speicher als JavaScript. Dies liegt am wesentlichen Unterscheid in den Ansätzen zum Speichermanagement zwischen JavaScript und WebAssembly. JavaScript verwendet eine Garbage Collection, ein dynamisches System zur Überwachung von Speicherzuweisungen, um festzustellen, wann nicht mehr benötigter Speicher freigegeben werden kann. Dieses Verfahren ermöglicht eine effiziente Nutzung des Speichers, indem es automatisch Bereiche identifiziert, die keine Verwendung mehr finden, und diese für neue Daten frei macht. WebAssembly hingegen verwendet ein lineares Speichermodell und reserviert für seine gesamte Laufzeit diesen linearen Speicher. Das führt dazu, dass die Größe des Programminputs die Leistung von WebAssembly erheblich beeinflusst. Bei der Verarbeitung kleiner bis mittelgroßer Datenmengen kann WebAssembly eine signifikante Leistungssteigerung gegenüber JavaScript aufweisen, mit potenziellen Geschwindigkeitsvorteilen von bis zu dem 26-fachen. Bei sehr großen Inputs kann jedoch JavaScript effizienter werden. Außerdem verursacht der Kontextwechsel zwischen JavaScript und WebAssembly zusätzlichen Laufzeit-Overhead.

#### Beeinflussung durch den Browser

Die Leistung von WebAssembly und JavaScript variiert erheblich, abhängig von der Ausführungsumgebung. Verschiedene JavaScript Engines, die in den jeweiligen Browsern zum Einsatz kommen, haben unterschiedliche Ausführungsgeschwindigkeiten und Speicherverbrauch für JavaScript und WebAssembly. Es ergaben sich über die 41 durchgeführten Test-Benchmarks folgende durchschnittliche Ergebnisse auf den drei verschiedenen Browsern:

Chrome (Desktop):

1. JavaScript Ausführungszeit: 45,57 ms
2. WebAssembly Ausführungszeit: 65,23 ms
3. JavaScript Speicherverbrauch: 885,10 KB
4. WebAssembly Speicherverbrauch: 2999,63 KB

Firefox (Desktop):

1. JavaScript Ausführungszeit: 48,26 ms
2. WebAssembly Ausführungszeit: 39,65 ms
3. JavaScript Speicherverbrauch: 505,41 KB
4. WebAssembly Speicherverbrauch: 2493,02 KB

Edge (Desktop):

1. JavaScript Ausführungszeit: 63,62 ms
2. WebAssembly Ausführungszeit: 83,53 ms
3. JavaScript Speicherverbrauch: 871,27 KB
4. WebAssembly Speicherverbrauch: 2996,20 KB

#### Fazit

Die Ergebnisse dieser Untersuchung verdeutlichen, dass das ausgewählte Architekturmodell für die Verwaltung des Speichers einzelner Threads einen beträchtlichen Einfluss auf die Leistung der WebAssembly-Module ausüben kann. Das Aktorenmodell könnte zu einer beschleunigten Performance von WebAssembly-Modulen beitragen, da sie mit weniger Speicher initialisiert werden müssen. Darüber hinaus sollte die Kompilierung nicht ausschließlich mit einem LLVM-Compiler in Erwägung gezogen werden; vielmehr sollten alternative Methoden ebenfalls Berücksichtigung finden. Dies gilt insbesondere, da der LLVM-Compiler nicht in der Lage ist, WebAssembly in Bezug auf die Instanziierung von WebWorkern zu optimieren, da ihm diese Funktionalität unbekannt ist.

### Bounds-Checking

Ein weiteres bedeutsames Paper, das auf der Studie "Understanding the Performance of WebAssembly Applications" aufbaut, ist das im Februar 2023 von Szewczyk et al. veröffentlichte Paper "Leaps and bounds: analysing WebAssembly's performance with a focus on bounds checking" (Szewczyk, Stonehouse, Barbalace, & Spink, 2023). Dieses Papier identifiziert, dass ein signifikanter Faktor für den Performanceverlust bei WebAssembly-Modulen in der Dauer des Bounds Checking Verfahren liegt. Bounds Checking bezeichnet das Verfahren, welches sicherstellt, dass Operationen auf linearem Speicher innerhalb seiner definierten Grenzen bleiben. Dieser Prozess verhindert, dass Speicher außerhalb dieser festgelegten Bereiche modifiziert wird. Diese Sicherheitsmechanismen können einen Overhead von bis zu 650% verursachen.

#### Fazit

Diese Untersuchung hebt hervor, wie kritisch Bounds Checking für die Gewährleistung der Speichersicherheit in WebAssembly ist, während es gleichzeitig signifikante Herausforderungen für die Performance darstellt. Speziell im Multithreading-Kontext, wo Threads gleichzeitig auf gemeinsame Speicherbereiche zugreifen, kann das Bounds Checking zu erheblichen Leistungseinbußen führen.

### Implementierungsansätze

In der Diskussion über aktuelle Implementierungsansätze für das Zusammenspiel von WebAssembly mit Multithreading, basierend auf den Erkenntnissen aus zwei führenden Büchern, lassen sich mehrere zentrale Themen identifizieren. Diese Analyse stützt sich auf "WebAssembly in Action" von Gerard Gallant, veröffentlicht im November 2019 (Gallant, WebAssembly in Action, 2019), und "Multithreaded JavaScript: Concurrency Beyond the Event Loop" von Thomas Hunter & Bryan English, veröffentlicht im Oktober 2021 (Hunter & English, Multithreaded JavaScript, 2021). Obwohl beide Werke unterschiedliche Perspektiven auf die Herausforderungen bieten, liefern sie komplementäre Einsichten in die effektive Nutzung von Multithreading in WebAssembly.

Die Autoren beider Bücher sind sich einig, dass Web Worker aufgrund ihrer hohen Startkosten und des erhöhten Speicherbedarfes, der daraus resultiert, dass jeder Web Worker seinen eigenes JavaScript Kontext besitzt, nicht dafür geeignet sind oft und großen Mengen initialisiert zu werden. Vielmehr sollte man die Web Worker als langlebige Prozesse in das System einbetten (Hunter & English, Webassembly, 2021) (Gallant, Threading: Web workers an pthreads, 2019).

#### P-Threads mittels Emscripten

Zudem erläutern beide Autoren die Nutzung von PThreads, die durch Emscripten unterstützt werden. Emscripten transpiliert PThreads im Hintergrund zu Web Workern und verwendet den im Transpilierungsprozess erzeugten JavaScript Glue-Code als Schnittstelle, um die erforderlichen API-Aufrufe zu tätigen, da Web Worker nicht direkt aus WebAssembly heraus instanziiert werden können. Die Implementierung von PThreads impliziert automatisch die Verwendung des geteilten Speichermodells für das Speichermanagement, da dieser Ansatz in C basierend auf diesem Modell entwickelt wurde. Weiterhin thematisieren die Autoren zusätzliche Möglichkeiten zur Thread-Instanziierung, die während der Transpilierung mit Emscripten ausgewählt werden können. Emscripten bietet dazu verschiedene spezifische Kompilierungsflags an (Hunter & English, Webassembly, 2021) (Gallant, Threading: Web workers an pthreads, 2019).

Die Flags, die zum Einsatz kommen können, sind:

* USE\_PTHREADS : Um die Unterstützung für pthreads zu aktivieren, was zur Anpassung des JavaScript-Glue Codes führt, welche wiederum die Generierung von WebWorkern ermöglicht.
* PTHREAD\_POOL\_SIZE : Bei der Initialisierung des Moduls wird durch dieses Flag ein Thread-Pool angelegt, der eine spezifizierte Anzahl an Web Workern erzeugt, die anschließend vom Modul verwendet werden können. Die Erstellung von mehr Web Workern als benötigt, führt zu einer Verschwendung von Prozesszeit bei der Modulinitialisierung sowie von Speicherplatz für die nicht genutzten Web Worker. Wird kein Thread-Pool über das Flag spezifiziert, beginnt die Verarbeitung der Threads nicht unmittelbar, da das Modul zuerst den Glue-Code nutzen muss, um die Browser-API anzusprechen und die Threads nachträglich zu starten.

#### Ergänzende Ansätze

"Multithreaded JavaScript" erweitert die Diskussion um zusätzliche Programmiersprachen wie AssemblyScript. Die eigens erstellte Programmiersprache, welche sich nah an TypeScript orientiert, bietet eine weitere Methode zur Erstellung von WebAssembly-Codes. Obwohl dieser Ansatz die direkte Erstellung von Threads innerhalb des WebAssembly-Moduls nicht unterstützt, ermöglicht er dennoch die vollständige Nutzung von Multithreading, indem Threads im JavaScript-Kontext erzeugt werden und Atomics unterstützt werden. Dadurch werden die Paradigmen "Aktoren" und "shared memory" in AssemblyScript realisierbar (Hunter & English, Webassembly, 2021).

"WebAssembly in Action" stellt die Verwendung der emscripten\_worker API vor, die es ermöglicht, WebWorker mittels Glue-Codes zu initiieren. Diese führen einen standardmäßigen WebWorker mit JavaScript-Code aus, der jedoch in der Lage ist, WebAssembly-Module zu laden. Die Kommunikation und Speicherteilung orientiert sich hierbei am Aktorenmodell, da die post\_message-Schnittstelle durch die spezifische emscripten-Funktion emscripten\_call\_worker für den Datenaustausch verwendet wird (Gallant, Threading: Web workers an pthreads, 2019).

#### Fazit

Die Autoren beider Bücher bieten mit ihren ergänzenden Methoden weitere Optionen für Kommunikation und Speichermanagement an. Sie präsentieren unterschiedliche Ansätze und betonen, dass das Modell des „shared memorys“ besonders für Szenarien geeignet ist, die einen intensiven Zugriff auf geteilte Datenstrukturen erfordern. Das Aktorenmodell hingegen wird für Aufgaben vorgeschlagen, die eine deutliche Abgrenzung vom Hauptthread verlangen. Jedoch liefert keines der beiden Werke konkrete Benchmarks, die aufzeigen, unter welchen Umständen welcher Multithreading-Ansatz in Verbindung mit WebAssembly vorzuziehen ist – weder für das Aktorenmodell noch für das „shared memory“ Modell. Zudem werden keine Schwellenwerte genannt, die angeben, ab welcher Datenmenge ein Modell leistungsfähiger ist.

Außerdem erläutern die Autoren verschiedene Ansätze für die einzelnen Modelle. Das „shared memory“ Modell lässt sich auf zwei Weisen realisieren:

* Durch Verwendung von PThreads und Emscripten.
* Durch das Bearbeiten des Multithreadings im JavaScript-Code.

Für das Aktorenmodell ergeben sich ebenfalls mehrere Implementierungsmöglichkeiten:

* Durch Nachrichtenhandling aufseiten von JavaScript
* Durch Nachrichtenhandling mittels der Emscripten-Api aufseiten von WebAssembly

Leider gehen die Autoren zwar auf diese Ansätze ein, jedoch ohne sich detailliert zu den spezifischen Vor- und Nachteilen zu äußern. Angaben über Leistungsunterschiede zwischen den verschiedenen Ansätzen fehlen ebenfalls. Die Unsicherheit in diesem Bereich wird durch die Anmerkung im Buch „Multithreaded JavaScript“ unterstrichen, dass sich die Praktiken und Methoden zur Erstellung von multithreaded WebAssembly-Code noch in der Entwicklung befinden, was darauf zurückzuführen ist, dass sich dieses Feld erst in jüngster Zeit entwickelt hat und vor Kurzem begann, wissenschaftliche Aufmerksamkeit auf sich zu ziehen (Hunter & English, Webassembly, 2021).

### Analyse der Multithreading-Strategien

In der Studie „Actors vs Shared Memory: two models at work on Big Data application frameworks“ von Silvia Crafa und Luca Tronchin wird die Wirksamkeit von zwei verbreiteten Modellen zur Interaktion von Nebenläufigkeit und Speicher untersucht: das Shared-Memory-Modell und das Aktorenmodell. Diese werden im Kontext der Big-Data-Analytik evaluiert, wobei konkrete Implementierungen nicht nur auf verschiedenen Threads, sondern über Cluster-Frameworks auf unterschiedlichen Rechnern getestet werden. Beide Modelle wurden mit den Programmiermodellen MapReduce und Bulk Synchronous Parallel (BSP) erprobt, die zur Verarbeitung großer Datenmengen in verteilten Systemen eingesetzt werden. MapReduce ermöglicht die Verteilung rechenintensiver Aufgaben über viele Server, was zu einer erheblichen Beschleunigung der Datenverarbeitung führt. BSP zeigt sich besonders wirksam bei Problemen, die regelmäßige Synchronisierung erfordern, und findet oft Anwendung in wissenschaftlichen Kontexten, die komplexe iterative Algorithmen beinhalten, wie etwa bei der Simulation physikalischer Systeme oder der Verarbeitung von Graphen. Experimentelle Ergebnisse zeigen, dass das Shared-Memory-Modell eine bessere Leistung bei der MapReduce-Implementierung bietet, was durch die sequenziellen Verarbeitungsphasen und den minimalen Synchronisationsbedarf begünstigt wird. Im Gegensatz dazu fördert das Aktorenmodell durch seinen asynchronen Nachrichtenaustausch und geringere Blockierungsanforderungen effizient die skalierbare Ausführung von BSP, das intensive Speichersynchronisationen erfordert (Silvia & Tronchin, 2015).

#### Fazit

Die Ergebnisse des Papiers betonen die Wichtigkeit der richtigen Modellauswahl für die Interaktion von Nebenläufigkeit und Speicher, basierend auf den spezifischen Anforderungen der Anwendungen. Es wurde festgestellt, dass das Aktorenmodell für Anwendungen geeignet ist, die einen hohen Grad an Synchronisation erfordern, während das Shared-Memory-Modell besser für Anwendungen mit umfangreicher Datenverarbeitung und minimalem Synchronisationsbedarf passt. Die Ergebnisse beziehen sich auf Prozesse, die auf verschiedenen Servern komplett getrennt ablaufen, mit Aufgaben und Daten, die verteilt und verarbeitet werden müssen. In dieser Arbeit wird untersucht, ob die Vorteile der beiden Modelle unter der Annahme, dass nur unterschiedliche Threads und teilweise derselbe Speicherbereich verwendet werden, ebenfalls zutreffen. Diese Arbeit wird daher aufklären, ob die Vorteile der beiden Modelle unter der spezifischen Bedingung der Nutzung verschiedener Threads im gleichen Speicherbereich bestehen bleiben.

# Methodik

Die vorliegende Arbeit verwendet einen quantitativen empirischen Forschungsansatz, um die zentrale Fragestellung zu untersuchen:

**"Welche Ansätze und Techniken ermöglichen die effizienteste Nutzung paralleler Verarbeitung in Verbindung mit WebAssembly zur Maximierung der Web-Performance?"**

## ****Implementierungsansätze****

Ziel ist es, durch experimentelle Untersuchungen und vergleichende Analysen herauszufinden, welche Kombinationen aus folgenden Elementen die beste Performance für parallele Datenverarbeitungsaufgaben im Web bieten:

* Modelle zur Interaktion von Nebenläufigkeit und Speicher:
  + Aktorenmodell
  + Shared-Memory-Modell
* WebAssembly Multithreading-Konzepte:
  + Web Worker Wasm Initialisierung (WWWI)
  + Wasm P-Thread Initialisierung (WPTI)

Abbildung 14 verdeutlicht den Unterschied zwischen den Modellen zur Interaktion von Nebenläufigkeit und Speicher, speziell in unserem Anwendungsfall. Das Aktorenmodell setzt auf eine Architektur des verteilten Speichers, bei der einzelne Threads relevante Daten in ihrem eigenen Speicherbereich verwalten und nur kleinere Aufgaben abarbeiten. Die Zusammenführung der Daten und Verteilung der Aufgaben werden den Aktoren überlassen. Im Gegensatz dazu nutzt das Shared Memory-Modell einen zentralen Speicherbereich, in dem alle Threads eigenständig ihre Aufgaben bearbeiten und die Ergebnisse über atomare Operationen in den gemeinsam genutzten Speicher schreiben.

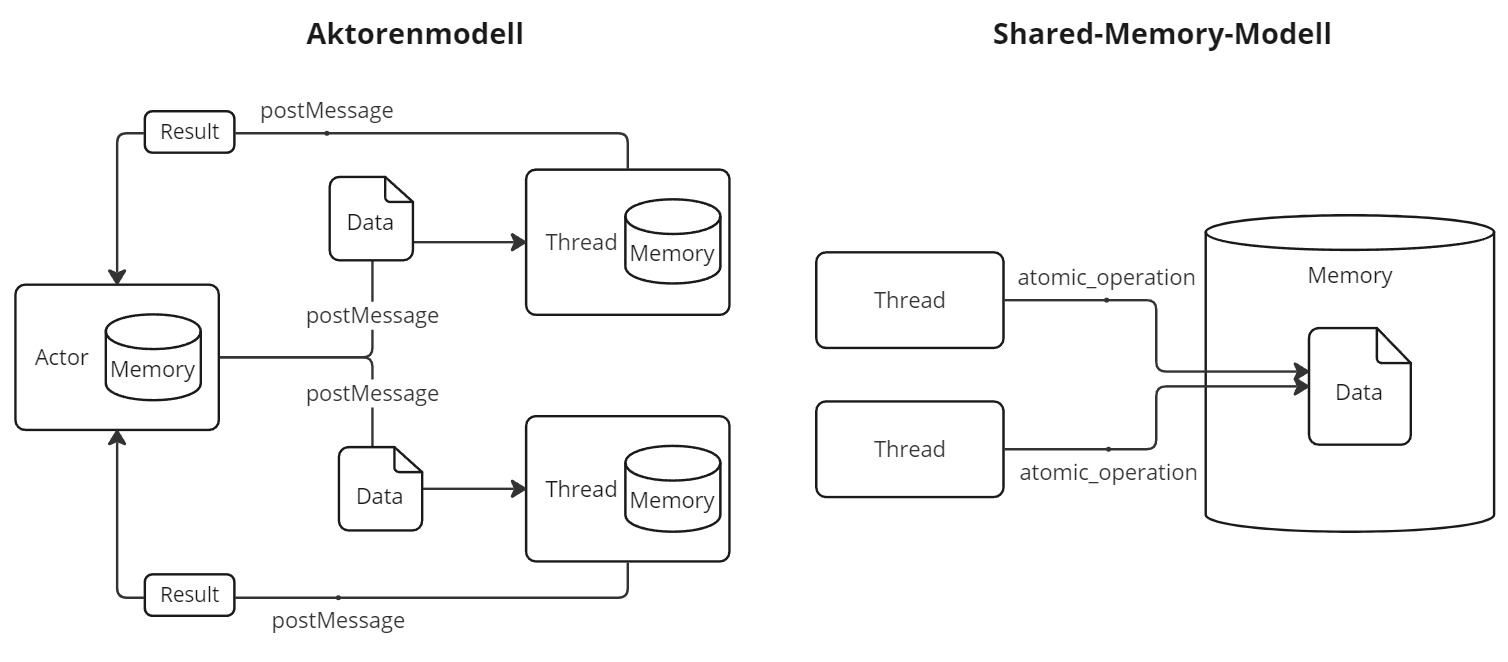


Abbildung 14: Modelle zur Interaktion von Nebenläufigkeit und Speicher (eigene Darstellung)

Abbildung 15 zeigt die Unterschiede der WebAssembly Multithreading-Konzepte in unserem Anwendungfall auf. Beim WWWI-Konzept werden Web Worker initialisiert, die die Instanzen der Wasm-Module initialisieren, während bei dem WPTI-Konzept ein Wasm-Modul initialisiert wird, das P-Threads verwendet. Diese nutzen intern durch die Emscripten-API Web Worker.

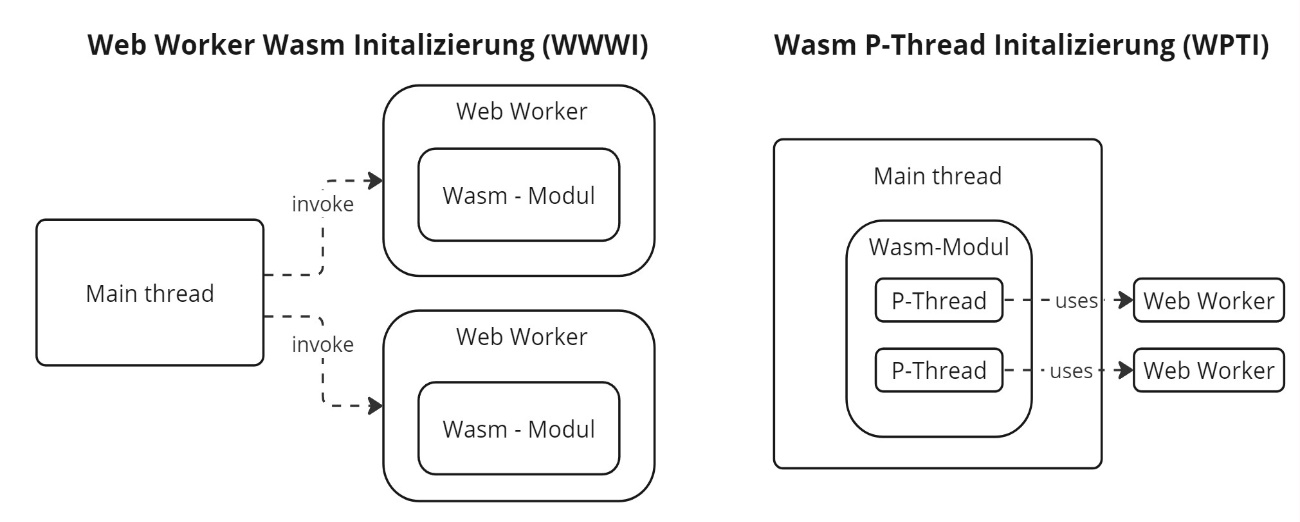


Abbildung 15: WebAssembly Multithreading-Konzepte (eigene Darstellung)

**Für jeden Implementierungsansatz kann ein Modell zur Interaktion zwischen Nebenläufigkeit und Speicher ausgewählt werden, sowie ein Konzept für Multithreading in WebAssembly. Tabelle X zeigt die Ansätze, die sich daraus ergeben. Die Namen der Implementierungsansätze werden im Code und in den Testauswertungen konsistent verwendet. Sie folgen einem einheitlichen Schema: [Technologie, die für die Lösung des Problems zuständig ist] + '\_' + [Modell zur Interaktion von Nebenläufigkeit und Speicher] + '\_' + [Technologie, die für das Multithreading verwendet wird].**

|  |  |  |
| --- | --- | --- |
|  | **WWWI** | **WPTI** |
| **Aktorenmodell** | **wasm\_actor\_webWorkers** | **wasm\_actor\_pthreads** |
| **Shared-Memory-Modell** | **wasm\_sharedMemory\_webWorkers** | **wasm\_sharedMemory\_pthreads** |

**Zusätzlich wurden weitere Implementierungsvarianten entwickelt, um den Performance-Gewinn in ein Verhältnis setzen und besser einordnen zu können. Die zusätzlichen Implementierungsansätze umfassen:**

* js\_no\_multithreading**: Ein Ansatz, der das Problem ohne Multithreading in JavaScript löst – quasi der Standard, der im Web verwendet wird.**
* wasm\_no\_multithreading**: Ein Ansatz, der das Problem ohne Multithreading in einem WebAssembly-Modul löst.**
* js\_actor\_webWorkers**: Eine reine JavaScript-Implementierung, die das Aktorenmodell verwendet und Web Workers als einzige verfügbare Option für Multithreading in JavaScript nutzt.**
* js\_sharedMemory\_webWorkers**: Eine reine JavaScript-Implementierung, die das Shared-Memory-Modell verwendet und ebenfalls Web Workers als einzige verfügbare Option für Multithreading in JavaScript einsetzt.**

## ****Das gewählte Leistungsproblem****

Im Rahmen dieser Forschung wurde eine Test-Suite entwickelt, die den DSatur-Algorithmus zur Knotenfärbung verwendet, um die Leistungsfähigkeit verschiedener Implementierungsansätze zu bewerten.

Dieser Algorithmus wurde speziell ausgewählt, da Knotenfärbung ein NP-schweres Problem darstellt, das potenziell hohe Rechenkapazitäten erfordert. Diese Eigenschaft macht es ideal für die Untersuchung der Verteilung der Lösung auf mehrere Threads. Der DSatur-Algorithmus bietet durch die Notwendigkeit, eine zusätzliche Variable pro Knoten zu synchronisieren, eine zusätzliche Herausforderung in Bezug auf die Synchronisierung des Speichers zwischen den Threads. Zudem priorisiert der DSatur-Algorithmus stets die Auswahl von Knoten zur Kolorierung, die an die größtmögliche Anzahl bereits gefärbter Knoten angrenzen. Dies verstärkt die Notwendigkeit der Koordination zwischen den Threads und macht die Speichersynchronisation zu einem kritischen Faktor, der explizit in unserem Szenario untersucht wird. Obwohl der DSatur-Algorithmus spezifisch für diese Studie gewählt wurde, sind die daraus gewonnenen Erkenntnisse auf eine Vielzahl ähnlich strukturierter Probleme übertragbar.

### ****Kategorisierung der Leistungsprobleme****

Im Laufe dieser Forschung wurden zwei Haupttypen von rechenintensiven Problemen identifiziert:

* **Probleme durch hohe Quantität:** Diese Kategorie umfasst Probleme, die durch die umfangreiche Menge der zu verarbeitenden Daten charakterisiert sind. Ein Beispiel hierfür ist der DSatur-Algorithmus bei einer großen Anzahl von Knoten und Kanten. Diese Probleme erfordern wegen ihrer hochfrequenten Speicherzugriffe eine intensive Synchronisierung des Speichers zwischen den einzelnen Threads, um Konsistenz und Datenintegrität zu gewährleisten.
* **Intrinsisch rechenintensive Probleme:** Im Gegensatz zu den quantitätsbasierten Problemen zeichnen sich diese durch die hohe Rechenintensität jeder einzelnen Aufgabe aus. Bei diesen Problemen ist die Frequenz der Speichersynchronisationen zwischen den Threads geringer, da weniger Interaktionen über gemeinsam genutzte Ressourcen stattfinden. Der Schwerpunkt liegt hier auf der Bewältigung der komplexen Berechnungen, die pro Aufgabe erforderlich sind, sodass die Herausforderung weniger in der Datenmenge als in der Komplexität der einzelnen Berechnungen liegt.

### ****Integration der Leibniz-Reihe zur Steuerung der Rechenintensität****

Um auch den zweiten Problemtyp abzudecken, wurde innerhalb des DSatur-Algorithmus eine zusätzliche Berechnung integriert: Jedes Mal, wenn ein Knoten eingefärbt wird, wird die Leibniz-Reihe zur Approximation der Kreiszahl Pi berechnet. Der Grad der Genauigkeit und damit die Rechenintensität der Operation werden durch die Anzahl der berechneten Terme der Reihe bestimmt. Dies ermöglicht eine Anpassung der Rechenlast pro Knotenfärbung, abhängig von der Anzahl der angegebenen Terme.

## ****Parameterkonfigurationen****

Daraus ergeben sich zwei Parameterkonfigurationen für einen Test, die die Art des Problems festlegen:

* Anzahl der Knoten und Kanten (Parametername: Nodes), mit den Werten:
  + 10
  + 100
  + 1000
* Anzahl der Terme in der Leibniz-Reihe (Parametername: Terms), mit den Werten:
  + 0
  + 100000

Zusätzlich zu den Hauptparametern werden zwei weitere Parameter durch die Testbedingungen bestimmt:

* Der verwendete Browser (Parametername: Browser), mit folgenden Optionen:
  + Google Chrome
  + Firefox
* Die Anzahl der Threads, die zur Problemlösung eingesetzt werden (Parametername: Threads), mit diesen möglichen Einstellungen:
  + 1
  + 2
  + 4
  + 8
  + 16
  + 32

Die Parameter wurden gezielt ausgewählt, um eine breite Palette rechenintensiver Szenarien abzudecken und die Effektivität verschiedener Implementierungsansätze umfassend zu evaluieren. Die Anzahl der Knoten und Kanten wurde so festgelegt, um die verschiedenen Größenordnungen der Probleme darzustellen, von relativ einfachen bis hin zu hochkomplexen Netzwerkstrukturen. Dies ermöglicht eine differenzierte Analyse der Leistungsfähigkeit unter variierenden Bedingungen der Datenquantität. Die Anzahl der Terme in der Leibniz-Reihe spiegelt die Extremwerte der Rechenintensität wider. Die Auswahl der Threadanzahl basiert auf der Überlegung, gängige Konfigurationen moderner Prozessoren zu simulieren.

## Erhobene Performance Metriken

In diesem Abschnitt werden die Metriken beschrieben, die zur Bewertung der Performance der verschiedenen Implementierungsansätze erhoben wurden. Diese Metriken sind entscheidend, um quantitative Daten zur Unterstützung unserer Analyse zu liefern und die Effektivität der verschiedenen getesteten Lösungen objektiv zu vergleichen.

Drei Hauptmetriken wurden im Rahmen dieser Studie verwendet:

* **Laufzeit in Millisekunden (ms):** Die Laufzeit ist die grundlegende Metrik zur Bewertung der Performance. Sie misst die Zeit, die benötigt wird, um ein definiertes Leistungsproblem zu lösen. Diese Metrik spiegelt die Effizienz des Implementierungsansatzes wider.
* **Gini-Koeffizient:** Dieser Koeffizient wird genutzt, um die Verteilung der Aufgaben über die genutzten Threads zu analysieren. Ein höherer Gini-Koeffizient weist auf eine ungleichere Verteilung zwischen den Threads hin und deutet auf Ineffizienzen in der Nutzung von Ressourcen hin.
* **Gesamtgröße des JavaScript-Heaps:** Diese Metrik gibt die gesamte Größe des von einer Implementierung verwendeten Speichers im JavaScript-Heap an. Sie dient als Indikator für den Speicherverbrauch und ist hilfreich, um Speicherlecks oder andere speicherbezogene Performanceprobleme zu identifizieren.

## Erfassung der Daten

Die Datenerfassung erfolgte automatisiert durch eine Test-Suite, die sicherstellt, dass alle Tests unter identischen Bedingungen durchgeführt werden. Jeder Test wurde mithilfe von Webpack separat gebündelt, um eine Isolation der Tests zu gewährleisten und gegenseitige Beeinflussungen zu vermeiden. Um den Einfluss der grafischen Benutzeroberfläche (GUI) auf die Leistungsmessung zu minimieren, wurde eine Option implementiert, die die Visualisierung innerhalb der Graphen deaktiviert. Diese Maßnahme wurde ergriffen, da die Aktualisierungen der GUI nicht direkt mit den Kernfunktionen der Tests zusammenhängen und lediglich zusätzliches Rauschen in den Datensätzen verursachen würden.

Die Kommunikation mit dem Hauptthread zur Übermittlung der Ergebnisse wurde dennoch erfasst. Es wurde zudem darauf geachtet, dass Ladevorgänge der GUI oder anderweitige Codeabschnitte nicht in die gemessene Testzeit einfließen. Das Laden des WebAssembly-Moduls wurde jedoch in die Testzeit einbezogen, da sich die Ladezeiten je nach Modell aufgrund unterschiedlicher Ladestrategien und möglicher doppelter Ladevorgänge in separaten Speicherbereichen unterscheiden können.

Nach Abschluss der Tests wurden die Daten zur zentralen Test-Suite zurückgesendet und dort in einer zentralen Datei gespeichert. Für die Auswertung der Daten wurden diese gruppiert und in Abhängigkeit von der Anzahl der verwendeten Threads analysiert. Die X-Achse repräsentiert hierbei die Anzahl der Threads, während auf der Y-Achse verschiedene erfasste Leistungsmetriken dargestellt werden, um vergleichende Analysen der Datensätze zu ermöglichen.

## Abgrenzung

Vor der Diskussion der Architektur der Performance-Test-Suite und der zugehörigen Testfälle ist es wichtig klarzustellen, welche Bereiche nicht entwickelt bzw. getestet werden. Diese Abgrenzung zielt darauf ab, den Fokus der vorliegenden Arbeit präzise zu definieren und zu verfolgen. Zugleich soll dem Leser ein Überblick über die Bereiche gegeben werden, in denen das untersuchte Problem auftritt, die jeweils einer separaten Analyse bedürfen. Hiermit wird explizit dargelegt, für welche Szenarien unsere Architekturen, Implementierungen und Ergebnisse Gültigkeit besitzen.

**WebAssembly-Schnittstellen**

Diese Arbeit beschränkt sich ausschließlich auf die Betrachtung von WebAssembly-Code, der innerhalb eines Browsers verarbeitet wird und somit nur auf die Web-API zugreift. Obwohl die Nutzung von WebAssembly in einer Node.js-Umgebung möglich ist – wobei dort das WebAssembly System Interface (WASI) genutzt wird –, wird dieser Anwendungsfall hier nicht behandelt. Die Verwendung von WASI führt zu signifikanten Unterschieden in der Interaktion von WebAssembly-Modulen mit ihrer Umgebung und im Zugriff auf Ressourcen. Während die Browserschnittstelle WebAssembly in einer webbasierten, durch JavaScript vermittelten Umgebung positioniert, ermöglicht WASI eine direktere, systemnahe Schnittstelle für die Ausführung von WebAssembly außerhalb des Webkontextes. Besonders betroffen von dieser Unterscheidung ist die Initialisierung von Web Workern, die in Node.js über eine gänzlich andere Schnittstelle abgewickelt wird.

**Browserspezifische Auswirkungen**

Wie bereits dargestellt, kann die ausgewählte Browser Engine die Performance von WebAssembly Modulen beeinflussen. Der Fokus der vorliegenden Arbeit liegt jedoch primär auf der Identifizierung des performantesten Ansatzes zur Implementierung von WebAssembly unter Berücksichtigung von Multithreading, unabhängig von spezifischen Browser-Engines. Auch wenn die Leistungsmessungen in verschiedenen populären Browsern betrachtet werden, strebt diese Untersuchung an, eine generell optimale Implementierungsstrategie zu erarbeiten, die browserübergreifend effizient ist. Dieser Ansatz wird gewählt, da der Endbenutzer den Browser bestimmt und Entwickler somit gefordert sind, eine für alle Browser optimierte Lösung zu implementieren. Zwar werden Performance-Unterschiede zwischen den Browsern erkennbar sein, doch zielt die empfohlene Implementierung darauf ab, in jedem Browser die bestmögliche Leistung zu erzielen.

**Erweiterter Vergleich von WebAssembly-Compilern**

WebAssembly kann aus einer Vielzahl von Sprachen transpiliert werden, wobei diverse Compiler zum Einsatz kommen. Einige dieser Compiler verfügen über eigene interne Mechanismen, um Threading, das in der Originalsprache implementiert wurde, auf WebWorker-basiertes Threading zu übertragen. Andere Compiler hingegen bieten keine Möglichkeit, das Multithreading der Ursprungssprache zu portieren. Die vorliegende Studie untersucht, ob portiertes Multithreading innerhalb eines WebAssembly-Moduls leistungsfähiger ist, als wenn das Multithreading-Management aus dem Modul extrahiert und in JavaScript gehandhabt wird. Um eine Antwort auf diese spezifische Problemstellung zu erhalten, genügt es, einen Compiler zu verwenden, dessen Vorteile auch in anderen Kontexten Anwendung finden können. Ein umfassender Test der einzelnen Compiler bezüglich der Multithreading-Portierung wird nicht durchgeführt.

**Schwellwerte für den Einsatz von Multithreading**

Während unserer Evaluation werden wir auf Fälle stoßen, in denen ein Ansatz ohne Multithreading performanter ist als einer mit Multithreading. Trotzdem werden solche Tests durchgeführt, um Startkosten der Threads und Verzögerungen zu erfassen. Wir werden jedoch keine Schwellwerte definieren, die bestimmen, ab wann der Einsatz von Multithreading sinnvoll ist. In unserem spezifischen Testumfeld wäre das wenig zielführend, da wir dadurch nur Schwellwerte ermitteln würden, die ausschließlich für unser spezielles Problem gelten. Unsere Ergebnisse sind vielmehr darauf ausgerichtet, Anwendungen, die sich bereits für den Einsatz von Multithreading bewährt haben, hinsichtlich ihrer Performance weiter zu optimieren.

# Architektur

In diesem Kapitel wird die Architektur der einzelnen Implementierungsansätze beschrieben. Ziel ist es, ein Verständnis für die technischen Grundlagen und den strukturellen Aufbau dieser Komponenten zu vermitteln.

## wasm\_sharedMemory\_pthreads

Der Implementierungsansatz wasm\_sharedMemory\_pthreads stellt den aktuellen Standard dar, um Multithreading in Kombination mit WebAssembly zu realisieren. Dieser Ansatz wird bevorzugt, weil viele Projekte lediglich ihre bestehenden Codebasen auf das Web übertragen wollen und dabei WebAssembly nutzen. Diese Projekte verfügen in der Regel bereits über eine etablierte Multithreading-Architektur, die meist aus Umgebungen wie C++ oder Rust stammt, wo das Shared-Memory-Modell der übliche Ansatz für Multithreading ist. In diesen Sprachen ist der Speicher standardmäßig für alle Threads zugänglich.

Abbildung 16 verdeutlicht die Architektur dieses Implementierungsansatzes, mit einem Fokus auf die Speicherbereiche. Der Haupt-Thread in JavaScript verfügt über einen eigenen Heap und Stack. Innerhalb seines Heap Speichers initialisiert er einen Shared Array Buffer, der als zentraler Speicher für das Wasm-Modul dient. Nach dem Laden des Wasm-Moduls übergibt der Haupt-Thread eine Referenz auf diesen geteilten Speicherbereich an das Modul. Das Wasm-Modul initialisiert daraufhin diesen Bereich als seinen zentralen Speicher, in dem es seinen eigenen Heap und Stack verwaltet. Während des Initialisierungsvorgangs des Wasm-Moduls wird zudem ein Thread-Pool vordefinierter Größe gestartet, der eine bestimmte Anzahl an WebWorkern bereitstellt. Die P-Threads, die im Modul gestartet werden nutzen intern diese WebWorker, um parallele Verarbeitungen zu ermöglichen. Die Graphdaten, die im Wasm-Heap gespeichert sind, können durch Atomic Operations sicher von den P-Threads bzw. WebWorkern modifiziert werden. Die Details zur internen Koordination der Pthreads durch die WebWorker werden durch Emscripten gesteuert und sind nicht einsehbar.

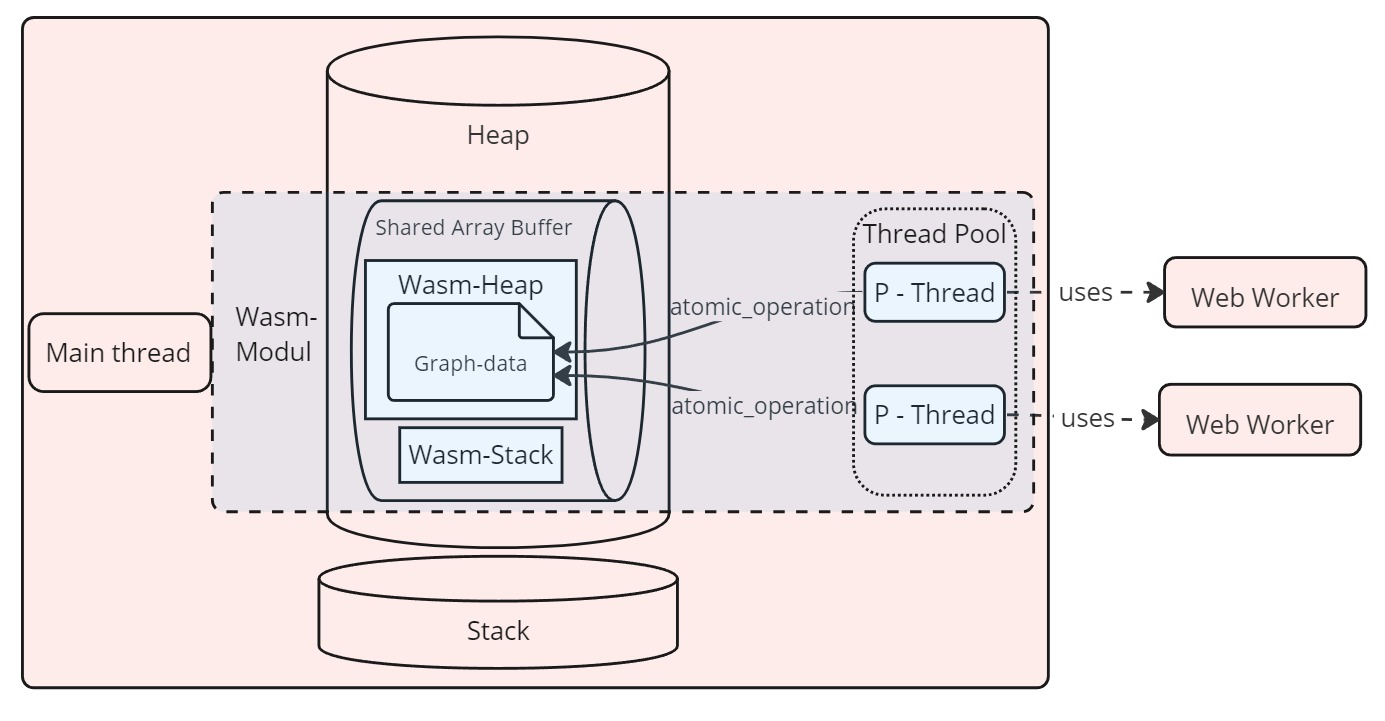


Abbildung 16: Architektur der Implementierung wasm\_sharedMemory\_pthread (eigene Darstellung)

## wasm\_sharedMemory\_webWorkers

Wählt man den alternativen Konzeptansatz für das Multithreading im Shared Memory Modell, so startet man eine bestimmte Anzahl an WebWorkern, die jeweils eine eigene Instanz des Wasm-Moduls verwalten. Dieser Ansatz findet unter Entwicklern wenig Verbreitung, wie eine Recherche in Entwicklerforen zeigt, in denen nur zwei relevante Beiträge zu diesem Thema gefunden wurden. In einem Beitrag berichtet ein Entwickler von überlappenden Stacks der Wasm-Module bei diesem Ansatz, was zur Entscheidung führte, die P-Thread-Variante zu implementieren (allan-bonadio, 2023). Der andere Beitrag erwähnt, dass P-Threads nicht geeignet sind, um in einem bestimmten Szenario große Datenmengen schnell zwischen Wasm-Modulen zu transferieren, sodass versucht wird, das Multithreading über WebWorker zu implementieren. Auch in diesem Beitrag wird keine praktikable Lösung für diesen Ansatz gefunden (frank-pian, 2023).

Abbildung 17 visualisiert die Architektur dieses Implementierungsansatzes. Im Heap des Hauptthreads wird ein Shared Array Buffer bereitgestellt, auf den eine definierte Anzahl von WebWorkern zugreift, denen eine Referenz auf den Shared Array Buffer übergeben wird. Diese WebWorker laden und initialisieren das Wasm-Modul unabhängig voneinander, indem sie denselben Shared Array Buffer für ihren Speicherbereich verwenden. Beide Module existieren in unterschiedlichen Kontexten und sind abgesehen von ihrem geteilten Speicher voneinander isoliert. Dadurch initialisieren sie den gesamten bereitgestellten Speicher für sich. Ohne zusätzliche Maßnahmen führt dies zu dem von einem Entwickler beschriebenen Verhalten: Die Wasm-Stacks überlappen sich. Diese Überlappung kann dazu führen, dass Funktionen oder Variablen eines Moduls den Stack-Bereich des anderen überschreiben, was zu unvorhersehbarem Verhalten führt. In unserem spezifischen Szenario führte dies zu fehlerhaften Speicherzugriffen. Ohne vorherige Maßnahmen ist dieser Implementierungsansatz unbrauchbar.

Um diesen Ansatz dennoch umsetzbar zu machen, wurde ein spezieller Prozess bei der Vorkompilierung der Module angewendet, um die Stack-Bereiche zu trennen. Dies erfordert, dass unser ursprüngliches Modul in verschiedenen Versionen kompiliert wird. Ein Modul wurde so kompiliert, dass es einen vordefinierten Stackbereich von nur einem MB nutzt, während das andere Modul mit der gleichen Funktionalität einen Stackbereich von zwei MB verwendet. Da die Stackbereiche von oben nach unten zugewiesen werden, tritt eine Überlappung nur dann auf, wenn das zweite Modul den 1-MB-Bereich überschreitet. Diese technische Anpassung ermöglicht die Nutzung des Ansatzes, erfordert jedoch zusätzliche Sicherheitsmechanismen, um die Speicherbereiche effektiv zu trennen. Durch die Notwendigkeit, für jeden weiteren WebWorker im Thread Pool ein zusätzlich angepasstes Wasm-Modul zu kompilieren, das die passende Stackgröße aufweist, beschränken wir uns in dieser Forschung darauf, diesen Implementierungsansatz mit höchstens zwei Threads auszuführen. Die Wasm-Module sind von der Multithreading-Logik entbunden, da diese auf der Ebene von JavaScript gehandhabt wird, und müssen lediglich ihre Daten mittels Atomic Operations manipulieren. So können simple Wasm-Module entwickelt werden, die dennoch komplexe Multithreading-Probleme lösen können.

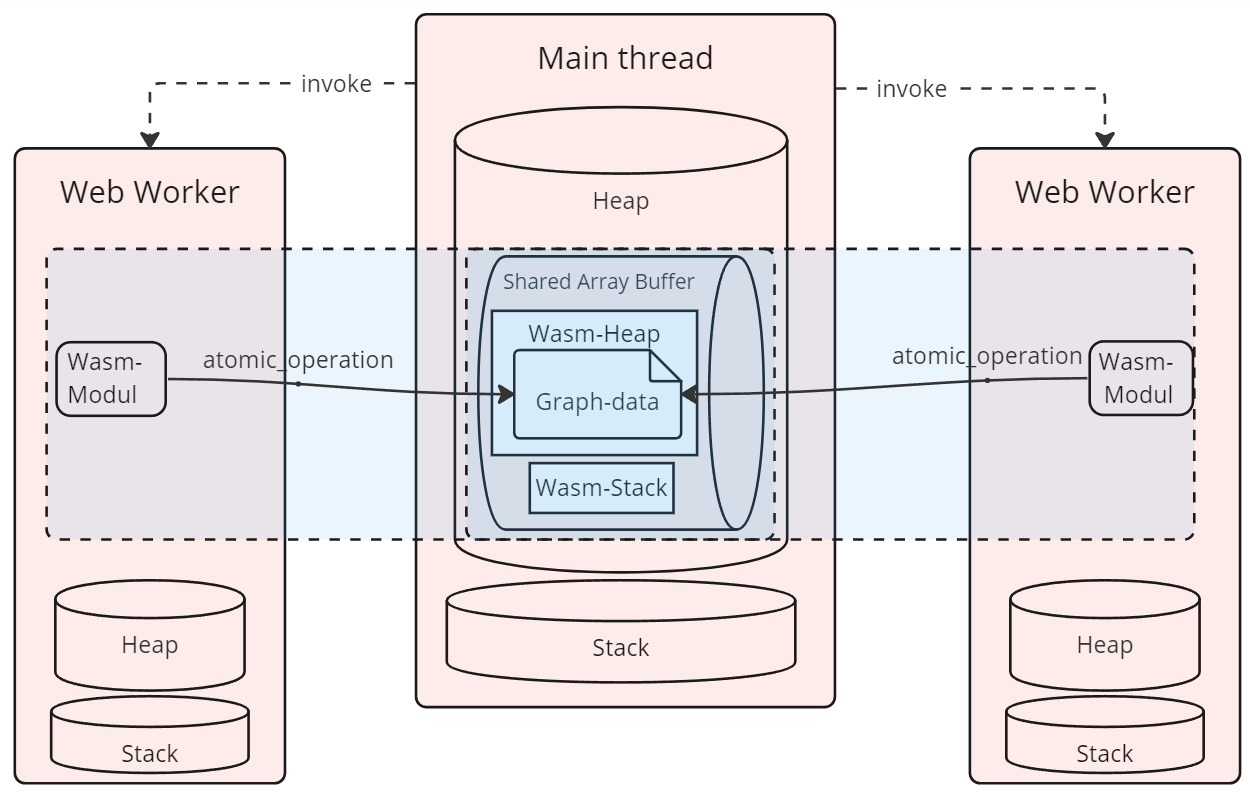


Abbildung 17: Architektur der Implementierung wasm\_sharedMemory\_webWorkers (eigene Darstellung)

## wasm\_actor\_webWorker

Der Implementierungsansatz wasm\_actor\_webWorker ist der einzige Ansatz, der ohne geteilten Speicher auskommt. Die Implementierung ähnelt stark dem herkömmlichen Vorgehen, um Multithreading in einer JavaScript-Umgebung zu nutzen, mit dem Unterschied, dass die Aufgabenbearbeitung im Wasm-Modul erfolgt.

Abbildung 18 illustriert den Architekturansatz mit einem Fokus auf die Verteilung der Aufgaben und die Speicherbereiche. Der Hauptthread agiert in unserem Fall als Akteur, der die WebWorker des Worker-Pools startet und Aufgaben an diese verteilt. In jedem WebWorker wird eine eigene Instanz des Wasm-Moduls initialisiert, die über ein LinearArrayBuffer verfügt. Dadurch sind die WebWorker, die Wasm-Module und deren Speicherbereiche vollständig voneinander isoliert. Die Graphdaten werden für jede Aufgabe, die im Wasm-Modul ausgeführt werden soll, neu mitgeschickt. Dies geschieht über die postMessage-Funktion des Akteurs. Entsprechend müssen die Daten des Graphen für jeden Kolorierungsvorgang eines Knotens vollständig geklont werden. Nachdem die Aufgabe im Wasm-Modul abgeschlossen ist, wird das Ergebnis über die postMessage-Funktion zurück an den Akteur gesendet. Dieser integriert alle Ergebnisse in seine Graphdaten und leitet dann die nächste Runde der Aufgabenverteilung ein. Da die Graphdaten voneinander isoliert sind, benötigen wir keine Synchronisationsmechanismen beim Schreiben von Daten. Nur der Akteur verändert aktiv die relevante Kopie der Graphdaten. Da dieser rein sequenziell arbeitet, müssen keine Synchronisationsmechanismen bei der Manipulation der Daten verwendet werden.

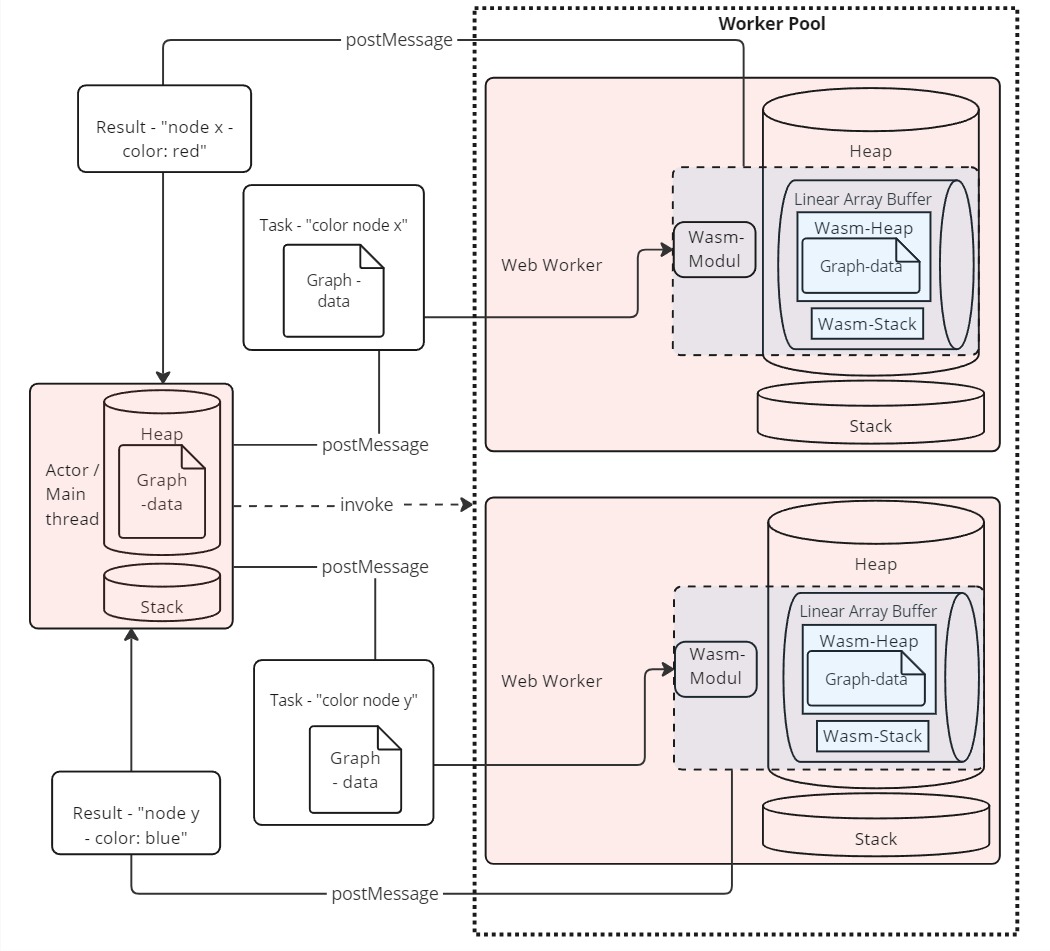


Abbildung 18: Architektur der Implementierung wasm\_actor\_webWorker (eigene Darstellung)

## wasm\_actor\_pthreads

Die Implementierung des Aktorenmodells innerhalb des Wasm-Moduls stellte eine Herausforderung dar, da die P-Threads innerhalb des Moduls nativ Zugriff auf die Datenstrukturen des Aktors besitzen. In der C++-Community existieren diverse Bibliotheken, die das Aktorenmodell implementieren und anbieten. Diese sind jedoch primär für den Einsatz mit nativem C++-Code konzipiert und ließen sich daher nicht unmittelbar auf unsere spezifische Implementierung des Aktorenmodells im Wasm-Kontext übertragen. Um die Testbedingungen konstant zu halten, wurde eine eigene Adaption des Aktorenmodells im C++-Code entwickelt.

Abbildung 19 zeigt die Architektur dieses Implementierungsansatzes. Der Hauptthread initialisiert das Wasm-Modul mit einem Shared Array Buffer, der notwendig ist, um P-Threads nutzen zu können, auch wenn wir in diesem Modell keinen geteilten Speicher benötigen würden. Das Wasm-Modul fungiert als Aktor und initialisiert beim Start einen Thread-Pool von WebWorkern, die später von den P-Threads genutzt werden. Die Aufgabenverteilung der Daten in diesem Modell wird anders gehandhabt als im reinen Aktorenmodell, da die P-Threads Zugriff auf den Hauptspeicher des Aktors haben und somit die Graphdaten nicht bei jedem Kolorierungsvorgang eines Knotens übergeben werden müssen. Es handelt sich um einen hybriden Ansatz, der das SharedArrayBuffer für die Datenübertragung nutzt, jedoch nur mit Leseoperationen, um keine Synchronisationsmechanismen für den Speicher verwenden zu müssen. Die Tasks werden über einen Funktionsaufruf an die P-Threads übergeben, und das Ergebnis wird dann als Rückgabewert an den Aktor geschickt, der dann sequenziell die Daten im Shared Array Buffer verändert. So kommen wir ohne Speichersynchronisierungsschritte aus, trotz der Nutzung des Shared Array Buffers.

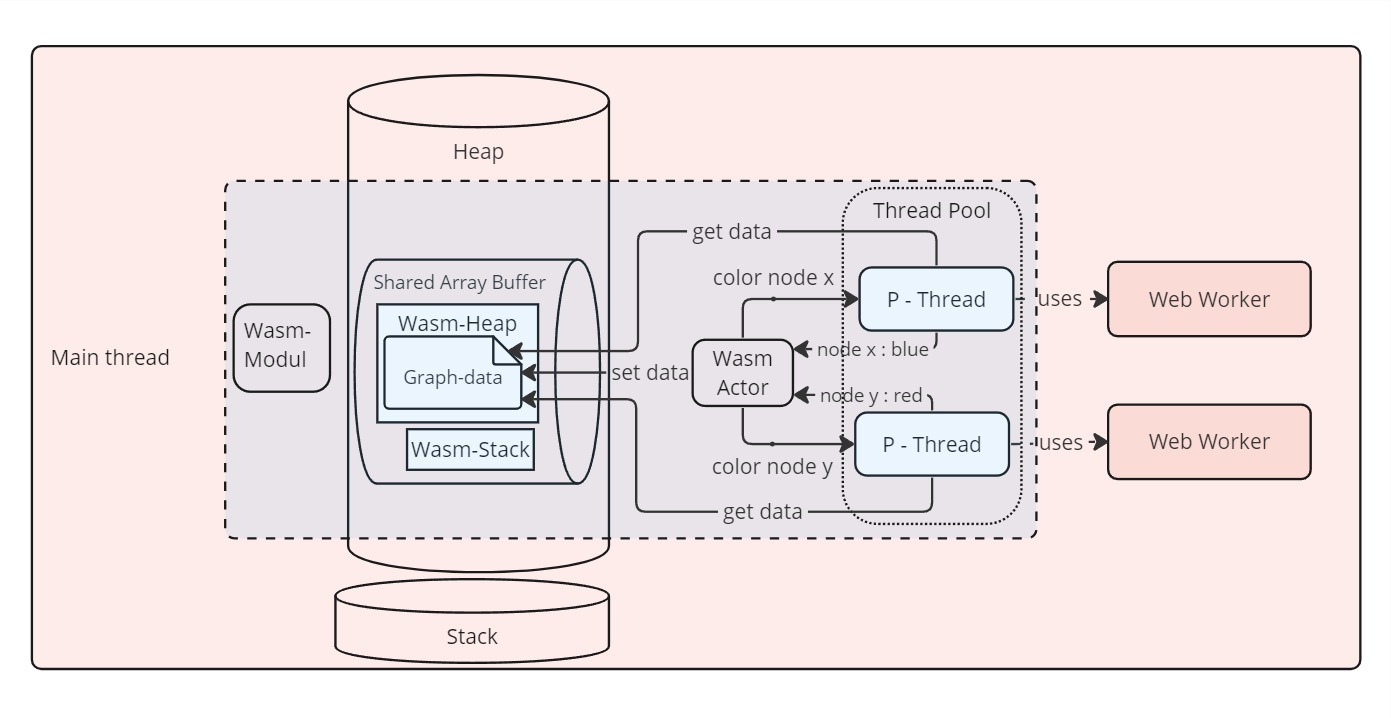


Abbildung 19: Architektur der Implementierung wasm\_actor\_pthread (eigene Darstellung)

# Implementierung

In diesem Kapitel untersuchen wir eingehend die Implementierung der Test-Suite, der Multithreading-Konzepte und der damit verbundenen Leistungsprobleme.

Der zugehörige Quellcode ist im folgenden Repository vollständig einsehbar: https://github.com/JonasStrt/performance\_tests\_multithreaded\_WebAssembly.

## Test-Suite

Die Test-Suite wird durch einen Node.js-Express-Server bereitgestellt, der neben der Bereitstellung einiger Wasm-Module über eine API auch die Testergebnisse zentral in einer JSON-Datei auf dem Server speichert. Beim Aufrufen der Webseite gelangt man auf eine Benutzeroberfläche, die in Abbildung 20 dargestellt ist. Hier können Benutzer die Parameter für Testdurchläufe festlegen und aus einer Liste die gewünschte Implementierung zur Graphenfärbung auswählen. Die Liste der Implementierungen ist umfangreicher als in der Methodik beschrieben, da Projekte, die P-Threads nutzen, mit einer festgelegten Thread-Pool-Größe kompiliert werden müssen. Dadurch ist es notwendig, die entsprechende Implementierung auszuwählen, um optimale Ergebnisse zu erzielen. Die Auswahl einer Implementierung mit einem zu großen oder zu kleinen Thread-Pool kann zu Ressourcenverschwendung bzw. Programmabstürzen führen, da nicht die geeignete Anzahl an P-Threads gestartet werden kann.

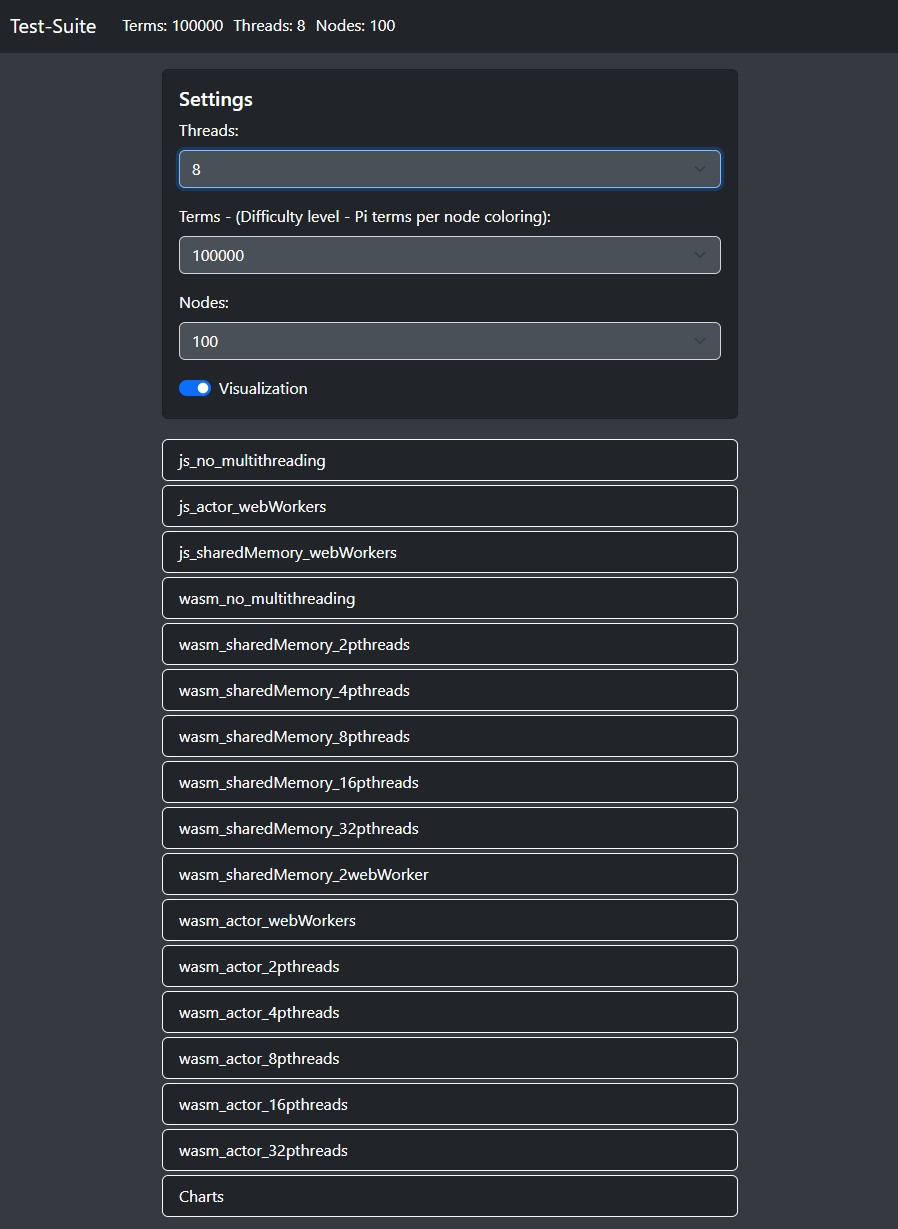


Abbildung 20: Startseite der Test-Suite

Nach Auswahl des Ansatzes und der Parameter wird eine Seite mit einem Diagramm und einem Bereich für die Messwerte geladen. Der Performance-Test färbt den angezeigten Graphen so, dass keine über eine Kante verbundenen Knoten dieselbe Farbe haben, während gleichzeitig die Anzahl der verwendeten Farben minimiert wird. Abbildung 21 illustriert die Ansicht des Tests nach der Durchführung der Färbung. Neben dem gefärbten Graphen und den erhobenen Messwerten wird auch angezeigt, welcher Thread wie viele Knoten gefärbt hat. Diese Werte müssen zusammen die Gesamtzahl der verwendeten Knoten ergeben; Abweichungen könnten auf fehlerhafte Speicherzugriffe hindeuten. In unseren implementierten Tests sind solche Fehlverhalten durch umfangreiche Optimierungen der Speichersynchronisation ausgeschlossen, sodass alle durch die Test-Suite erhobenen Tests als valide betrachtet werden können.

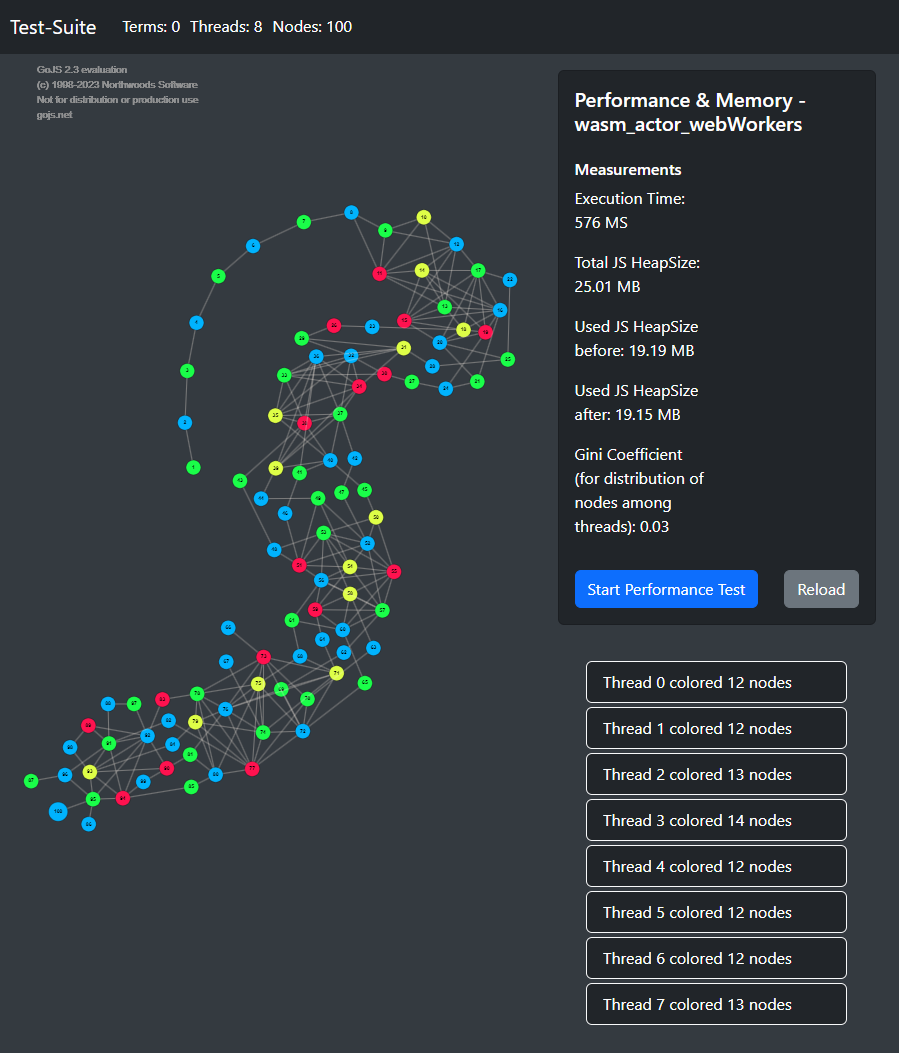


Abbildung 21: Ansicht eines Performance-Tests mit Graphenvisualisierung

Die Visualisierung des Graphen dient ausschließlich Evaluationszwecken, um das Verhalten der Algorithmen nachvollziehen zu können. In den analysierten Tests wurden keine Visualisierungen verwendet. Der Graph wird mittels der Bibliothek Go.js (GoJs, kein Datum) erstellt und weist stets dieselben Knoten und Kanten auf, obwohl die Knoten bei der Erstellung verschiedene Positionen zugewiesen bekommen können, was zu optisch unterschiedlichen Graphen führt. Diese Positionen sind für die Tests jedoch irrelevant. Der Graph wird durch einen deterministischen Algorithmus für eine vorgegebene Anzahl von Knoten generiert, mit dem Ziel, einen hochvernetzten Graphen zu erstellen, der eine Vielfalt an Vernetzungstypen aufweist. In dem Beispiel aus Abbildung 21 wird beispielsweise ein Knoten mit hoher Vernetzung, aber auch Knoten mit nur einer Kante dargestellt. Quellcode 1 präsentiert den implementierten Algorithmus, der innerhalb der Funktion generateGraph definiert ist, welche die Graphenerstellung steuert. Initial wird ein Array erstellt, das die gewünschte Anzahl an Knoten enthält. Anschließend erfolgt die systematische Generierung der Kanten. Nach einem bestimmten Intervall, das von der Größe des Graphen abhängt, wechselt der Algorithmus zwischen zwei Modi für die Kantenerstellung: Im Vortex-Modus werden Knoten mit anderen nahegelegenen Knoten im Array vernetzt, während im zyklischen Modus jeder Knoten mit einem weiter entfernten Knoten, abhängig vom linkIncrementFactor, verbunden wird. Nach einer bestimmten Anzahl von Knoten im aktuellen Modus wechselt der Modus und der linkIncrementFactor wird erhöht. So beginnen wir beispielsweise im zyklischen Modus und vernetzen jeden Knoten mit einem im Array direkt benachbarten Knoten. Nach 10 Knoten wechselt der Modus in den Vortex-Modus, in dem Knoten mit nahegelegenen Knoten im Array vernetzt werden. Nach weiteren 10 Knoten beginnen wir erneut mit dem zyklischen Modus, wobei nun der übernächste Knoten mit dem aktuellen Knoten verknüpft wird.

/\*\*

 \* Generiert einen Graphen mit einer bestimmten Anzahl von Knoten

 \* und verbindet diese durch Kanten.

 \* Die Verbindungen zwischen den Knoten wechseln zwischen einem

 \* zyklischen Modus und einem "Vortex"-Modus,

 \* abhängig von der Position im Graphen und einem Intervall

 \* basierend auf der Gesamtzahl der Knoten.

 \*

 \* @param {number} totalNodes -

 \* Die Gesamtzahl der Knoten im Graphen.

 \* @returns {Object}

 \* Ein Objekt, das zwei Arrays enthält: 'nodes' und 'links'.

 \* 'nodes' enthält Objekte, die einzelne Knoten repräsentieren.

 \* 'links' enthält Objekte, die Verbindungen

 \* zwischen den Knoten repräsentieren.

 \*/

function generateGraph(totalNodes) {

  let nodes = [];

  let links = [];

  let linkIncrementFactor = 1;

  let linkingInterval = totalNodes / 10;

  let isVortexModeActive = false;

  // Erstelle alle Knoten mit initialen Werten

  for (let nodeIndex = 1; nodeIndex <= totalNodes; nodeIndex++) {

    nodes.push({ key: nodeIndex, color: 0, weight: 0, lock: 0 });

  }

  // Überprüfe, ob eine ungerichtete Kanten zwischen zwei Knoten besteht

  function linkExists(source, target) {

    return links.some(

      (link) =>

        (link.from === source && link.to === target) ||

        (link.from === target && link.to === source)

    );

  }

  // Erstelle Kanten für jeden Knoten

  for (let nodeIndex = 1; nodeIndex <= totalNodes; nodeIndex++) {

    // Überprüfe, ob wir uns im Vortex Modus befinden

    if (isVortexModeActive) {

      // Erstelle wirbelartige Knonstrukte

      // indem jeder Knoten in diesem Modus

      // mit Knoten in seiner Nähe verbunden wird

      const possibleLinks = [1, 5, -3, -2];

      possibleLinks.forEach((offset) => {

        let targetNode = nodeIndex + offset;

        if (

          targetNode > 0 &&

          targetNode <= totalNodes &&

          !linkExists(nodeIndex, targetNode)

        ) {

          links.push({ from: nodeIndex, to: targetNode });

        }

      });

    } else {

      // Erstelle eine zyklische Struktur abhängig

      // von dem linkIncrementFactor

      // die Operation Modulo totalNodes stellt sicher,

      // dass die erstellte Kante innerhalb der Knoten ist

      let targetNode =

      ((nodeIndex + linkIncrementFactor - 1) % totalNodes) + 1;

      if (!linkExists(nodeIndex, targetNode)) {

        links.push({ from: nodeIndex, to: targetNode });

      }

    }

    // Nachdem eine Anzahl an Knoten in dem aktuellen Modus vernetzt

    // worden sind, ändere den Modus

    if (nodeIndex % linkingInterval === 0) {

      // Erhöhe den linkIncrementFactor,

      // sodass im nächsten zyklischen Modus

      // der Knoten mit einem Knotem + linkIncrementFactor

      // vernetzt wird

      linkIncrementFactor++;

      isVortexModeActive = !isVortexModeActive;

    }

  }

  return { nodes, links };

}

Quellcode 1: Funktion generateGraph

Zur Analyse der Ergebnisse wurde eine Benutzeroberfläche entwickelt, die es ermöglicht, die Daten in Diagrammform darzustellen. Für die Implementierung dieser Funktionalität kam die JavaScript-Bibliothek Chart.js (Chart.js, kein Datum) zum Einsatz. Abbildung X illustriert diese Oberfläche. Auf der rechten Seite befindet sich ein Auswahlmenü, über das sich die anzuzeigenden Implementierungsdaten und die zugehörige Performance-Metrik festlegen lassen. Die Kurve repräsentiert den Verlauf über verschiedene Threads. Auf der linken Seite wird das Diagramm mit den ausgewählten Werten visualisiert.

## Leistungsprobleme

Die untersuchten Leistungsprobleme umfassen den DSatur-Algorithmus zur Knotenfärbung eines Graphen sowie die Leibniz-Reihe zur Berechnung der Kreiszahl Pi. Diese Methoden wurden gezielt kombiniert, um die Auswirkungen von Multithreading in Verbindung mit WebAssembly auf verschiedene Leistungsszenarien evaluieren zu können.

Abbildung 22 zeigt ein vereinfachtes Aktivitätsdiagramm des DSatur-Algorithmus. Um die Übersichtlichkeit zu gewährleisten, wurden Hilfsfunktionen lediglich mit ihrem Namen aufgeführt; diese werden jedoch in folgenden Quellcode Abschnitten erläutert. Der vollständige Quellcode für die Funktion dSatur ist im Anhang unter ‚Quellcode-Anhang 1: Funktion dSatur‘ aufgeführt. In unserem Szenario besitzt ein Knoten folgende Attribute: Id, Color, Saturation und Locked. Die Id dient zur eindeutigen Identifikation des Knotens. Die Farbe (Color) wird später einem Hexadezimalwert zugeordnet, die Sättigung (Saturation) zeigt den Sättigungsgrad des Knotens an, und Sperrung (Locked) indiziert, ob der Knoten aktuell von einem anderen Thread bearbeitet wird. Zu Beginn der Funktionsausführung sind alle Attribute außer der Id auf 0 gesetzt.

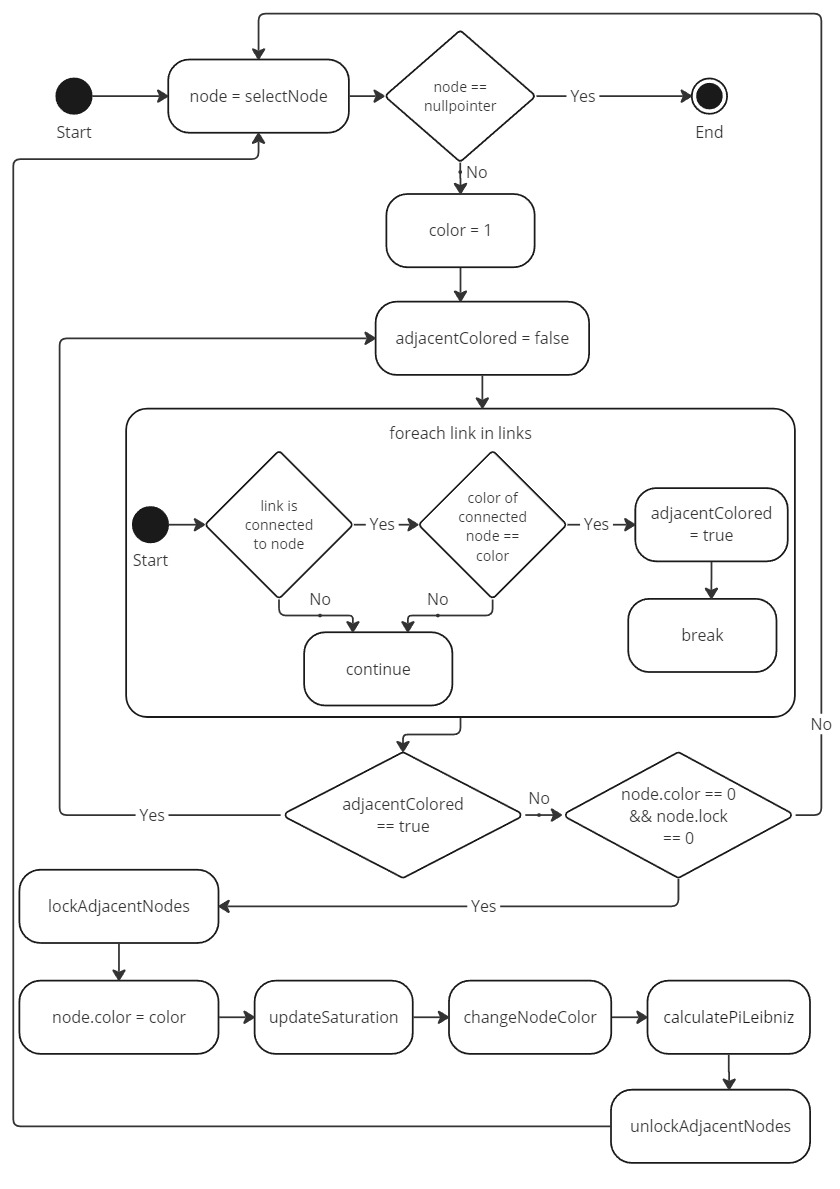


Abbildung 22: Aktivitätsdiagramm zur Visualisierung der implementierten DSatur-Funktion (eigene Darstellung)

Der Prozess beginnt mit der Auswahl eines Knotens. Quellcode 2 illustriert die Funktion selectNode, die hierfür verantwortlich ist. Die Funktion iteriert durch das Array der Knoten und wählt jeden Knoten aus, der noch nicht gefärbt und nicht gesperrt ist. Die Sättigung und der Grad der Verbindungen des Knotens werden gespeichert. Sollte die Sättigung die bisher höchste sein, wird dieser Knoten als potenzieller Kandidat für die nächste Färbung gespeichert. Bei einem Gleichstand der Sättigung gibt der Grad der angrenzenden Verbindungen den Ausschlag. Falls alle Knoten bereits gefärbt sind, gibt die Funktion einen Nullpointer zurück, und der Algorithmus kann beendet werden. Andernfalls beginnt die Farbauswahl.

/\*\*

 \* Wählt den am besten geeigneten Knoten für die Färbung

 \* aus einer Liste von Knoten aus.

 \* Die Auswahl basiert auf der Sättigung und der Anzahl

 \* der Verbindungen des Knotens.

 \*

 \* @param nodes Referenz auf einen Vektor von `Node`-Objekten,

 \* die die Knoten des Graphen repräsentieren.

 \* @param links Konstante Referenz auf einen Vektor von `Link`-Objekten,

 \* die die Verbindungen zwischen den Knoten repräsentieren.

 \* @return Zeiger auf den ausgewählten `Node` oder nullptr,

 \* falls kein geeigneter Knoten gefunden wird.

 \*

 \* Der Algorithmus durchläuft alle Knoten und wählt den Knoten

 \* mit der höchsten Sättigung aus.

 \* Bei einem Gleichstand der Sättigung wird der Knoten

 \* mit der höchsten Anzahl an Verbindungen gewählt.

 \* Ein Knoten wird nur ausgewählt,

 \* wenn er noch nicht gefärbt und nicht gesperrt ist.

 \*/

Node\* selectNode

(std::vector<Node> &nodes, const std::vector<Link> &links)

{

    // Initial null, da noch kein Knoten ausgewählt wurde

    Node \*nodeToColor = nullptr;

    // Startet mit dem kleinstmöglichen Int-Wert für die Sättigung

    int maxSaturation = std::numeric\_limits<int>::min();

    // Startet mit dem kleinstmöglichen Int-Wert für die Link-Anzahl

    int maxLinkCount = std::numeric\_limits<int>::min();

    // Iteriere durch alle Knoten,

    // um den besten Kandidaten für die Färbung zu finden

    for (auto &currNode : nodes)

    {

        // Überprüfe, ob der Knoten ungefärbt und nicht gesperrt ist

        if (currNode.color.load() == 0 && currNode.lock.load() == 0)

        {

            // Lade die Sättigung des aktuellen Knotens

            int saturation = currNode.saturation.load();

            // Zähle die Verbindungen für den aktuellen Knoten

            int linkCount = countLinksForNode(currNode.key, links);

            // Aktualisiere den Knoten zur Färbung,

            // wenn dieser Knoten ein höheres Gewicht hat

            // oder bei gleichem Gewicht mehr Verbindungen

            if (saturation > maxSaturation ||

               (saturation == maxSaturation

                && linkCount > maxLinkCount))

            {

                nodeToColor = &currNode;

                maxSaturation = saturation;

                maxLinkCount = linkCount;

            }

        }

    }

    // Gibt den ausgewählten Knoten zurück oder nullptr,

    // wenn kein passender gefunden wurde

    return nodeToColor;

}  
Quellcode 2: Funktion selectNode

Die Farbe wird zunächst mit dem ersten möglichen Wert initialisiert. Anschließend wird für jeden mit dem ausgewählten Knoten verbundenen Knoten überprüft, ob dessen Farbe mit der gewählten übereinstimmt. Falls ein angrenzender Knoten dieselbe Farbe aufweist, wird die Farbe erhöht, und die Iteration durch die Verbindungen beginnt von neuem. Sobald keine angrenzenden Knoten mehr dieselbe Farbe wie die ausgewählte aufweisen, kann zum nächsten Schritt übergegangen werden.

Falls kein anderer Thread den ausgewählten Knoten gesperrt oder gefärbt hat, wird der Knoten eingefärbt. Falls der Knoten bereits von einem anderen Thread bearbeitet wird, hat es sich als effizienter erwiesen, einen neuen Knoten auszuwählen, anstatt auf die Freigabe zu warten. Bevor der neue Farbwert im Knoten gespeichert wird, werden alle angrenzenden Knoten gesperrt (Funktion: lockAdjacentNodes, siehe Quellcode-Anhang 3), um eine Bearbeitung durch andere Threads zu verhindern. Danach wird der Farbwert für den Knoten im geteilten Speicher gespeichert und die Sättigung der angrenzenden Knoten aktualisiert (Funktion: updateSaturation, siehe Quellcode-Anhang 2). Ein Callback wird aufgerufen, der dem Hauptthread die neue Farbe des Knotens übermittelt, um sie in das Diagramm einzufärben, falls eine Visualisierung gewünscht ist.

Abschließend wird das nächste Leistungsproblem, die Leibnizreihe, bearbeitet. Die Funktion calculatePiLeibniz führt die entsprechende mathematische Reihe aus:

Quellcode 3 illustriert die Implementierung der Funktion calculatePiLeibniz in C++. Ein wesentliches Element dieser Implementierung ist die Verwendung des Schlüsselwortes volatile für die Variable sum. Dieses Schlüsselwort instruiert den Compiler, die Variable so zu behandeln, als könnte sie jederzeit durch die Hardware modifiziert werden. Diese Vorgehensweise ist erforderlich, um eine mögliche Eliminierung der Berechnungen durch Compiler-Optimierungen zu verhindern. Insbesondere könnte der Emscripten-Compiler beim Kompilierungsprozess erkennen, dass das Ergebnis der Funktion nicht verwendet wird und dementsprechend die Berechnungen eliminieren, die zu dieser Variablen führen. In diesem speziellen Fall soll die Berechnung allerdings künstlichen Overhead erzeugen, was sie zu einem potenziellen Ziel für eine derartige Optimierung macht. Durch die Deklaration der Variable als volatile wird dem Compiler signalisiert, dass die Variable potenziell jederzeit von der Hardware genutzt werden könnte, was eine “Optimierung“ verhindert. Zusätzlich wird das Ergebnis der Pi-Berechnung mit hoher Präzision ausgegeben. Diese Maßnahme ermöglicht die Überprüfung der Genauigkeit der berechneten Pi-Werte. Die Initialisierung der Variable als long double gewährleistet zudem ausreichend Speicherplatz für die präzise Darstellung der Gleitkommazahl. Die Umsetzung der Leibniz-Reihe erfolgt durch eine Schleife, die für jeden ungeraden Durchlauf einen Term von der Summe subtrahiert und für jeden geraden Durchlauf einen Term hinzufügt. Dieses Vorgehen spiegelt das mathematische Verhalten der alternierenden Serie wieder, wobei der alternierende Charakter durch die Verwendung des Schleifenindex i in der Berechnungsformel umgesetzt wird.

/\*\*

 \* Berechnet die Kreiszahl Pi mit der Leibniz-Reihe.

 \*

 \* Diese Funktion verwendet die alternierende

 \* Leibniz-Reihe zur Approximation von Pi:

 \* Pi = 4 \* (1 - 1/3 + 1/5 - 1/7 + 1/9 - ...)

 \*

 \* @param terms Anzahl der Terme der Reihe,

 \* die für die Berechnung von Pi verwendet werden sollen.

 \* @return Die durch die Leibniz-Reihe

 \* berechnete Approximation von Pi.

 \*

 \* Die Genauigkeit der Approximation steigt mit der Anzahl

 \* der berücksichtigten Terme.

 \* Die Funktion gibt außerdem die errechnete

 \* Pi-Approximation in der Konsole aus.

 \*/

double calculatePiLeibniz(int terms)

{

    // Verwendet für eine präzise Akkumulation der Terme

    volatile long double sum = 0.0;

    // Iteriere durch die Anzahl der angegebenen Terme

    for (int i = 0; i < terms; i++)

    {

        // Addiere oder subtrahiere den i-ten Term

        // basierend auf der Parität von i

        if (i % 2 == 0)

        {

            // Positive Terme der Reihe

            sum += 1.0 / (2 \* i + 1);

        }

        else

        {

            // Negative Terme der Reihe

            sum -= 1.0 / (2 \* i + 1);

        }

    }

    // Multipliziere die Summe mit 4, um Pi zu approximieren

    long double pi = 4 \* sum;

    // Ausgabe des Ergebnisses mit hoher Präzision

    std::cout << "Pi berechnet mit " << terms << " Termen: "

              << std::setprecision(20) << pi << std::endl;

    // Gibt die berechnete Pi-Approximation zurück

    return pi;

}

Quellcode 3: Funktion calculatePiLeibniz

Nach Abschluss dieser Berechnungen werden alle gesperrten angrenzenden Knoten wieder freigegeben (Funktion: unlockAdjacentNodes, siehe Quellcode-Anhang 4).

Das Aktivitätsdiagramm ist für beide Modelle der Interaktion zwischen Nebenläufigkeit und Speichermanagement anwendbar. Im Aktorenmodell werden spezifische Schritte — von der Initialisierung der Farbe bis zur Sperrung der angrenzenden Knoten — in Threads ausgeführt. Zusätzlich erfolgt die Berechnung der Leibniz-Reihe innerhalb eines Threads. Alle anderen Aktionen, wie die Auswahl der Knoten, das Sperren angrenzender Knoten, die Zuweisung der Farbe, der Aufruf des Callbacks zur Färbung des Diagramms und das Entsperren der angrenzenden Knoten, werden vom Aktor übernommen. Im Shared-Memory-Modell werden alle Schritte der DSatur-Funktion in Threads verarbeitet. Bei Implementierungen ohne Multithreading werden die Aktionen zur Sperrung anderer Knoten ausgelassen, und es erfolgt keine Prüfung, ob ein anderer Thread den ausgewählten Knoten bereits gefärbt hat.

Das im Aktivitätsdiagramm dargestellte Verfahren wird in nahezu jedem Implementierungsansatz durch C++-Code realisiert, der anschließend nach WebAssembly kompiliert wird. Eine Ausnahme bildet die Implementierung wasm\_actor\_webWorkers, bei der alle Funktionen, die innerhalb des Aktors ausgeführt werden, in JavaScript umgesetzt sind. Für Vergleichszwecke wurden die Ansätze js\_no\_multithreading, js\_actor\_webWorkers und js\_sharedMemory\_webWorkers vollständig in reinem JavaScript implementiert, diese stehen aber nicht im Hauptfokus unserer Analyse.

## Implementierungsansätze

In diesem Abschnitt erfolgt eine detaillierte Untersuchung der Implementierungsansätze. Die Analyse umfasst spezifische Herausforderungen und Besonderheiten, die während der Implementierung und Kompilierung zu den Wasm-Modulen auftraten. Zusätzlich werden die speziellen Maßnahmen beschrieben, die zur Initialisierung der Threads, zur Synchronisation des Speichers und zur Datenübermittlung an die Wasm-Module ergriffen wurden.

Die grundlegende Struktur der Implementierungsansätze ist so konzipiert, dass das Skript diagramm.js als Einstiegspunkt für einen Test dient. Dieses Skript erstellt die Graphdaten, initialisiert das go.js-Diagramm, startet den Performance-Test und erfasst dabei Laufzeit sowie Speicherverbrauch. Die Werte werden abschließend zum Server gesendet. Die Laufzeitmessung erfolgt über die WebAPI performance, einem Standardverfahren, das auch in den im Kapitel 'Verwandte Arbeiten' analysierten Publikationen zur Messung der Ausführungszeit verwendet wird. Quellcode 4 illustriert die Messmethode für die Laufzeit in unseren Performance-Tests.

  const start = performance.now();

  //start test

  await startTest(nodes, links, terms, threads);

  //end test

  const end = performance.now();

  const time = end - start;

Quellcode 4: Quellcode zur Messung der Laufzeit

Der Gini-Koeffizient wird in der Funktion calculateGiniCoefficient berechnet, welche im Quellcode-Anhang 5 dargestellt ist.

In der vorliegenden Studie wird der Speicherverbrauch mittels totalJSHeapSize aus der Web-API performance.memory erfasst. Diese Größe ist jedoch nicht völlig verlässlich, da sie stark von der Ausführung der Garbage Collection abhängt, welche die Genauigkeit beeinträchtigen kann. Studien haben gezeigt, dass die zeitliche Koordination und das Ausmaß der Speicherfreigabe durch die Garbage Collection erhebliche Schwankungen in den gemessenen Werten verursachen können, wodurch diese Methode für präzise Messungen ungeeignet ist (Kra, 2020). Außerdem ist diese API aktuell nur im Chrome Browser nutzbar. Daher dient der erfasste Speicherverbrauch in dieser Analyse lediglich als grober Richtwert, während der Fokus hauptsächlich auf der Messung der Laufzeit liegt. Quellcode 5 illustriert die methodische Vorgehensweise zur Erfassung der Heap-Größe in JavaScript, die zur Bewertung der Speichernutzung implementiert wurde.

const memoryAfter = window.performance.memory;

const totalJsHeapSize =

    (memoryAfter.totalJSHeapSize / 1048576).toFixed(2);

Quellcode 5: Erfassung der Heap-Größe in JavaScript

Es wird ausschließlich die Laufzeit der Funktion startTest erfasst, die ein JavaScript-Promise zurückgibt, sobald alle Knoten koloriert sind. Die Funktion startTest wird aus dem JavaScript-Skript test.js importiert, welches die relevante Arbeitslast, die gemessen werden soll, bereitstellt. Dieses Skript ist verantwortlich für die Serialisierung der Graphdaten, das Starten der Threads, die Initialisierung der Wasm-Module und die Ausführung der Funktion processGraph innerhalb der Wasm-Module. Die Funktion processGraph ist dafür zuständig, die serialisierten Daten entgegenzunehmen und sie mithilfe des dSatur-Algorithmus zu verarbeiten. In den folgenden Abschnitten werden die spezifischen Implementierungen für die einzelnen Ansätze erläutert.

### wasm\_sharedMemory\_pthreads

Die Implementierung des Ansatzes wasm\_sharedMemory\_pthread ist unkompliziert, da die Funktion processGraph im JavaScript-Hauptthread ausgeführt wird. Dadurch steht das Wasm-Modul direkt zur Verfügung, da es beim Laden der Test-Suite dem Hauptthread zugeordnet wird. Im Gegensatz dazu führen andere Module die Funktion processGraph aus einem separaten Thread aus, wodurch das Wasm-Modul in den neuen Kontext geladen werden muss.

In der Funktion startTest, die in Quellcode 6 dargestellt wird, werden folgende Schritte durchgeführt:

* Promise-Initialisierung:
  + Ein Promise wird erstellt, um das Ende des Tests zu signalisieren.
* Berechnung der Puffergrößen:
  + Node Buffer: 16 Bytes pro Knoten (4 Werte à 4 Bytes: id, color, saturation, lock).
  + Link Buffer: 8 Bytes pro Verbindung (2 Werte à 4 Bytes: from, to).
* Initialisierung des SharedArrayBuffer:
  + Ein SharedArrayBuffer wird basierend auf der kombinierten Puffergröße für Knoten und Kanten erstellt.
* Serialisierung der Graph Daten:
  + Die Funktion serializeGraph (Quellcode-Anhang 6) wird verwendet, um die Graph Daten in den SharedArrayBuffer zu serialisieren.
* Übertragung von JavaScript-Callbacks:
  + Die Funktion changeColor zur Farbänderung im Diagramm und threadsFinished, die das Promise auflöst, werden dem Wasm-Modul übergeben.
* Speicherreservierung:
  + Module.\_malloc reserviert Speicher im Wasm-Heap und gibt einen Pointer in Form eines Integerwerts zurück, der auf den Anfang des reservierten Speichers zeigt.
* Datenübertragung in den Wasm-Heap:
  + Die Graphdaten werden mithilfe von Module.HEAP32 an die reservierte Speicherstelle im Wasm-Heap übertragen. Die Division durch 4 ist notwendig, da HEAP32 32-Bit-Werte (4 Byte) speichert.
* Aufruf der processGraph-Funktion:
  + processGraph wird im Wasm-Modul aufgerufen, wobei die Pointer auf den reservierten Speicherbereich sowie die Längen von Knoten und Verbindungen übergeben werden. Zusätzlich werden die Terme für die Leibniz-Reihe und die Anzahl der zu initialisierenden Threads übergeben.
* Speicherfreigabe:
  + Abschließend wird auf das angelegte Promise gewartet, um danach den reservierten Speicher freizugeben.

Diese Vorgehensweise zur Übertragung von Daten an das Wasm-Modul ist die einzig praktikable, da nur einfache Datentypen und ArrayBuffer zwischen JavaScript und dem Wasm-Modul ausgetauscht werden können. Die Speicherung der Daten im Heap des Wasm-Moduls vor der Verarbeitung ist daher ein eleganter Ansatz.

/\*\*

 \* Führt die `startTest`-Funktion asynchron aus, die die Graphdaten

 \* für die Test-Suite vorbereitet und den dSatur-Algorithmus im

 \* WebAssembly-Modul ausführt.

 \*

 \* @param \_nodes Array der Knoten, die gefärbt werden sollen.

 \* @param \_links Array der Kanten, die die Verbindungen

 \*               zwischen den Knoten darstellen.

 \* @param \_terms Anzahl der Terme für die Berechnung der Leibniz-Reihe.

 \* @param \_threads Anzahl der Threads,

 \*                 die initialisiert werden sollen.

 \*/

async function startTest(\_nodes, \_links, \_terms, \_threads) {

  // Initialisiere das Promise, auf das gewartet wird,

  // wenn alle Knoten koloriert sind

  initializePromise();

  // Übernehme die Parameterwerte in globale Variablen

  terms = \_terms;

  threads = \_threads;

  nodes = \_nodes;

  links = \_links;

  // Bestimme die benötigte Puffergröße für die Knoten und Kanten

  // 16 Bytes pro Knoten (4 Werte à 4 Bytes)

  nodeBufferSize = nodes.length \* 16;

  // 8 Bytes pro Kante (2 Werte à 4 Bytes)

  linkBufferSize = links.length \* 8;

  totalBufferSize = nodeBufferSize + linkBufferSize;

  // Initialisiere den SharedArrayBuffer für die kombinierte Puffergröße

  buffer = new SharedArrayBuffer(totalBufferSize);

  int32View = new Int32Array(buffer);

  // Serialisiere die Graphdaten in den SharedArrayBuffer

  serializeGraph();

  // Stelle sicher, dass das WebAssembly-Modul verfügbar ist

  if (typeof Module !== "undefined") {

      // Übergibt die Callback-Funktionen für die Farbänderung

      // und den Abschluss der Threads

      Module.changeNodeColor = changeNodeColor;

      Module.threadsFinished = threadsFinished;

      // Reserviere den benötigten Speicher im WebAssembly-Modul

      const nodesPtr = Module.\_malloc(totalBufferSize);

      // Übertrage die serialisierten Graphdaten in den Wasm-Heap

      Module.HEAP32.set(int32View, nodesPtr / 4);

      // Bestimme die Positionen der Knoten- und Kantenarrays im Speicher

      const nodeCount = nodes.length;

      const linkCount = links.length;

      const linksPtr = nodesPtr + nodeBufferSize;

      // Starte den dSatur-Algorithmus mit den übergebenen Parametern

      Module.\_processGraph(

          nodesPtr,

          nodeCount,

          linksPtr,

          linkCount,

          terms,

          threads

      );

      // Warte auf den Abschluss der Threads,

      // bevor der Speicher freigegeben wird

      await threadsFinishedPromise;

      Module.\_free(nodesPtr);

  }

}

Quellcode 6: Funktion startTest

Um unseren C++-Code, der den dSatur-Algorithmus in mehreren Threads ausführt, mithilfe von Emscripten in WebAssembly zu kompilieren, wurde folgender Befehl (Quellcode 7: emscripten Befehl für wasm\_sharedMemory\_pthread) verwendet:

emcc dSaturWorker.cpp -o dSaturWorker.js --no-entry -s

EXPORTED\_FUNCTIONS='["\_malloc","\_free","\_processGraph"]' -s

USE\_PTHREADS=1 -s PTHREAD\_POOL\_SIZE=2

Quellcode 7: emscripten Befehl für wasm\_sharedMemory\_pthread

Die Bedeutung der einzelnen Befehlsoptionen ist in der folgenden Liste beschrieben:

* emcc: Der Emscripten-Compiler für C/C++-Code.
* dSaturWorker.cpp: Die Eingabedatei, die kompiliert wird.
* -o dSaturWorker.js: Gibt die Ausgabedatei an, die in diesem Fall das JavaScript-Frontend für WebAssembly enthält.
* --no-entry: Kein main-Einstiegspunkt erforderlich; wird genutzt, wenn Funktionen manuell aus JavaScript aufgerufen werden.
* -s EXPORTED\_FUNCTIONS='["\_malloc","\_free","\_processGraph"]':
  + Exportiert spezifische Funktionen, sodass sie im erzeugten WebAssembly-Modul über JavaScript zugänglich sind:
    - \_malloc: Reserviert Speicher.
    - \_free: Gibt Speicher frei.
    - \_processGraph: Die Hauptfunktion zum Verarbeiten des Graphen.
* -s USE\_PTHREADS=1: Aktiviert Pthreads, um Multithreading zu ermöglichen.
* -s PTHREAD\_POOL\_SIZE=2: Erstellt und reserviert einen Pthread-Pool mit zwei Threads für parallele Verarbeitung.

Ein weiteres potenziell nützliches Flag ist PROXY\_TO\_PTHREAD=1, welches das WebAssembly-Modul in einem P-Thread ausführt. In den aktuellen Tests wurde dieses Flag jedoch nicht genutzt, wodurch das WebAssembly-Modul den Hauptthread blockiert und visuelle Aktualisierungen erst nach Abschluss des Tests sichtbar werden. Die Nutzung dieses Flags würde eine grundlegende Anpassung der Testarchitektur erfordern, da die gesamte Verarbeitung in der main-Funktion des Wasm-Moduls erfolgen müsste, die direkt nach dem Laden der Test-Suite mit der Initialisierung des Moduls beginnt. Dies würde die Architektur dieses Moduls von den anderen Wasm-Modulen unterscheiden. Die Ausführung des gesamten Moduls in einem P-Thread hätte jedoch keine signifikanten Performance-Einbußen zur Folge. Da unsere Tests ohnehin ohne visuelle Anzeige durchgeführt werden und die Verarbeitung ausschließlich in den vom Wasm-Modul gestarteten Threads erfolgt, ist das Fehlen visueller Ausgaben für die Testergebnisse unerheblich. Daher wurde entschieden, die Architektur der WebAssembly-Module einheitlich zu gestalten, um eine konsistente und vergleichbare Testumgebung zu schaffen. Dennoch wird empfohlen, das Flag in anderen Kontexten zu nutzen, da es so den Hauptthread von JavaScript nicht blockiert und Visualisierungen unmittelbar angezeigt werden.

Die Funktion processGraph (Quellcode-Anhang 7: Funktion processGraph) in unserem Wasm-Modul ist so strukturiert, dass sie die Daten aus dem Heap deserialisiert und in einem std::vector der Typen Node und Link speichert. Die Datenstrukturen Node und Link sind in Quellcode 9 dargestellt. Die Attribute color, saturation und lock sind als std::atomic<int> deklariert, um mithilfe der atomic-Bibliothek sichere Speicherzugriffe zu gewährleisten.

/\*\*

 \* Struktur `Node`:

 \* Repräsentiert einen Knoten im Graphen mit atomaren Attributen

 \* zur sicheren Multithreading-Verarbeitung.

 \*/

struct Node {

    // Eindeutiger Identifikationsschlüssel des Knotens

    int key;

    // Aktuelle Farbe des Knotens

    std::atomic<int> color;

    // Sättigungsgrad oder Gewichtung des Knotens

    std::atomic<int> saturation;

    // Sperrstatus des Knotens

    std::atomic<int> lock;

    // Standard-Konstruktor

    Node() : key(0), color(0), saturation(0), lock(0) {}

    // Parametrisierter Konstruktor

    Node(int k, int c, int w, int l)

    : key(k), color(c), saturation(w), lock(l) {}

    // Kopierkonstruktor und Zuweisungsoperator

    // werden explizit gelöscht,

    // um unbeabsichtigtes Kopieren dieser Struktur zu vermeiden.

    Node(const Node &) = delete;

    Node &operator=(const Node &) = delete;

    // Move-Konstruktor:

    Node(Node &&other) noexcept

        : key(other.key),

          color(other.color.load()),

          saturation(other.saturation.load()),

          lock(other.lock.load()) {}

    // Move-Zuweisungsoperator

    Node &operator=(Node &&other) noexcept {

        key = other.key;

        color.store(other.color.load());

        saturation.store(other.saturation.load());

        lock.store(other.lock.load());

        return \*this;

    }

};

/\*\*

 \* Struktur `Link`:

 \* Stellt eine Verbindung zwischen zwei Knoten dar,

 \* definiert durch Start- und Endknoten.

 \*/

struct Link {

    int from; // Ursprungknoten der Verbindung

    int to;   // Zielknoten der Verbindung

};

Quellcode 8: Datenstrukturen - Links und Nodes

Anschließend startet die Funktion invokeThreads (Quellcode-Anhang 8) eine Anzahl von Threads entsprechend dem übergebenen Parameter threads. Jeder dieser Threads führt die dSatur-Funktion aus, der ein Zeiger auf die Graph-Daten übergeben wird. Nach Abschluss aller Threads wird das JavaScript-Callback aufgerufen, das den Test beendet.

### wasm\_sharedMemory\_webWorkers

Die Implementierung dieses Ansatzes ist, wie bereits im Architekturbereich beschrieben, deutlich komplexer. Das liegt daran, dass das Wasm-Modul außerhalb des Hauptkontextes verwendet wird und deshalb explizit pro WebWorker geladen werden muss. Darüber hinaus existiert nicht nur ein Wasm-Modul, sondern aufgrund der Probleme mit überlappenden Stack-Bereichen müssen zwei verschiedene Module geladen werden: eines mit einem 1-MB-Stack-Bereich für den ersten WebWorker-Thread und ein weiteres mit einem 2-MB-Stack-Bereich für den anderen WebWorker-Thread.

Um dies zu realisieren, wurde eine zusätzliche Klasse namens DSaturWasmModuleWrapper implementiert. Diese Klasse importiert das gewünschte Wasm-Modul (1-MB- oder 2-MB-Stack-Bereich), weist diesem ein übergebenes WebAssembly.Memory-Objekt zu und gibt ein Promise zurück, wenn das Modul vollständig geladen ist. Das WebAssembly.Memory-Objekt basiert auf einem SharedArrayBuffer, der es ermöglicht, dass die beiden Wasm-Modul-Instanzen, die in unterschiedlichen Kontexten laufen, ein gemeinsames Speicherobjekt teilen. Dies kann nur während der Initialisierung des Wasm-Moduls erfolgen und erfordert ein WebAssembly.Memory-Objekt. Letztendlich ist dies jedoch vergleichbar mit der direkten Arbeit mit einem SharedArrayBuffer, da das WebAssembly.Memory-Objekt intern auf einem solchen Buffer basiert.

In diesem Fall ist die JavaScript-Seite dafür verantwortlich, die WebWorker zu starten, was ein separates Skript erfordert, das innerhalb eines WebWorkers ausgeführt wird. Zu diesem Zweck wurde das Skript dSaturWasmWorker.js erstellt. Dieser WebWorker kann in zwei Modi gestartet werden: im Initialisierungsmodus für den Speicher und im Ausführungsmodus zur Ausführung der processGraph-Funktion im Wasm-Modul. In beiden Fällen lädt der WebWorker zuerst das Wasm-Modul über den DSaturWasmModuleWrapper.

Im Initialisierungsmodus wird das im Hauptthread erstellte WebAssembly.Memory-Objekt sowie die Graph Daten übergeben. Dieser Modus dient dazu, das geteilte WebAssembly.Memory-Objekt, das im Hauptkontext von JavaScript erstellt wurde, mit den Graph Daten zu füllen. Hierbei wird die gleiche Methode wie im ersten Ansatz verwendet: Der Speicher wird mit der exportierten malloc-Funktion des Wasm-Moduls reserviert, die Daten werden an die entsprechende Stelle im Heap geschrieben und die Pointer auf diese Adressen werden dem Hauptthread zurückgegeben.

Im Ausführungsmodus wird die processGraph-Funktion des Wasm-Moduls gestartet, indem ihr die Länge der Knoten und Kanten, die Anzahl der Threads, die Anzahl der Terme und die Pointer auf die Startadressen im Heap übergeben werden.

In der Funktion startTest für dieses Modul werden die Schritte, die sich von der vorherigen Implementierung unterscheiden, detailliert beschrieben, während identische Schritte nur als Oberpunkte aufgeführt werden:

* Erstellung des WebAssembly.Memory-Objekts:
  + Ein WebAssembly.Memory-Objekt wird erstellt, das als gemeinsamer Speicher für die isolierten Wasm-Module dient, die in separaten WebWorkern laufen. Das Speicherobjekt wird explizit über ein Flag als geteilt gekennzeichnet.
* Berechnung der Puffergrößen:
* Serialisierung der Graphdaten:
* Starten des dSaturWasmWorker im Initialisierungsmodus:
  + Der oben beschriebene dSaturWasmWorker wird im Initialisierungsmodus gestartet, um die Graphdaten in den SharedArrayBuffer zu schreiben. Ein Promise stellt sicher, dass der nächste Schritt erst ausgeführt wird, wenn der Hauptthread die Pointer von den WebWorkern erhalten hat.
* Erstellung und Ausführung der dSaturWasmWorker:
  + Abhängig von der angegebenen Anzahl der Threads werden (maximal zwei) dSaturWasmWorker im Ausführungsmodus gestartet und geben ein Promise zurück, sobald alle Knoten koloriert sind.
* Warten auf den Abschluss der Promises:
  + Warten Sie auf die Fertigstellung aller Promises, bevor der Test als abgeschlossen betrachtet wird.

Die beschriebenen WebAssembly-Module wurden mit Emscripten mit Hilfe der folgenden Befehle erstellt:

* Wasm-Modul mit einem 1-MB-Stack-Bereich:

emcc dSaturSharedMemory1MB.cpp -o

dSaturSharedMemory1MB.js -s

EXPORTED\_FUNCTIONS='["\_malloc","\_free","\_processGraph"]'

-s IMPORTED\_MEMORY

-s SHARED\_MEMORY=1

-s ALLOW\_MEMORY\_GROWTH=0

-s TOTAL\_STACK=1MB

--no-entry

* Wasm-Modul mit einem 2-MB-Stack-Bereich:

emcc dSaturSharedMemory2MB.cpp -o

dSaturSharedMemory2MB.js -s

EXPORTED\_FUNCTIONS='["\_malloc","\_free","\_processGraph"]'

-s IMPORTED\_MEMORY

-s SHARED\_MEMORY=1

-s ALLOW\_MEMORY\_GROWTH=0

-s TOTAL\_STACK=2MB

--no-entry

Die neu eingeführten Befehle, die im vorherigen Abschnitt nicht beschrieben wurden, haben die folgenden Auswirkungen:

* IMPORTED\_MEMORY: Ermöglicht die Nutzung eines importierten Speicherobjekts.
* ALLOW\_MEMORY\_GROWTH=0: Verhindert die automatische Vergrößerung des Speichers.
* TOTAL\_STACK: Legt die Stack-Größe für das Modul fest.

Die Funktion processGraph in unserem Wasm-Modul ist ähnlich strukturiert wie im vorherigen Ansatz, mit dem Unterschied, dass nach der Deserialisierung der Daten aus dem Heap keine P-Threads gestartet werden. Stattdessen initiiert das Wasm-Modul direkt die dSatur-Funktion mit den deserialisierten Daten.

Im Rahmen einer Umstrukturierung wurden zwei verschiedene Ansätze getestet, um die Daten zu verarbeiten und mittels atomarer Operationen zu manipulieren. Einerseits können die Daten, wie im vorherigen Ansatz, in Vektoren des Typs Node konvertiert werden, wobei die zu manipulierenden Werte als std::atomic<int> gespeichert sind. Andererseits können die Daten auch direkt im Speicher manipuliert werden. Hierfür wurden Hilfsfunktionen entwickelt, die die von Emscripten bereitgestellten Funktionen der emscripten\_atomic-Bibliothek nutzen, um Daten atomar zu verändern. Quellcode 9 zeigt eine solche Hilfsfunktion zum Setzen eines Wertes. Der Funktion wird die Knoten-ID, der Attribut-Index und der neue Wert übergeben. Anschließend wird der Index im Speicher berechnet, an dem sich der zu manipulierende Wert befindet. Mithilfe der Funktion emscripten\_atomic\_store\_u32 wird an dieser Stelle ein neuer Wert thread-sicher geschrieben.

/\*\*

 \* Setzt ein bestimmtes Attribut eines Knotens

 \* im globalen Knoten-Datenarray

 \* mithilfe atomarer Operationen.

 \*

 \* @param nodeKey Der eindeutige Schlüssel des Knotens,

 \*  dessen Attribut gesetzt werden soll.

 \* @param attributeIndex Der Index des zu setzenden Attributs

 \*  innerhalb des Knotens.

 \* @param value Der neue Wert,

 \*  der für das Attribut gesetzt wird.

 \*/

void setNodeAttribute(

    int nodeKey,

    int attributeIndex,

    int value)

    {

    // Berechne den Index des Attributs im globalen Knoten-Datenarray.

    // Jeder Knoten hat 4 Attribute, daher (nodeKey - 1) \* 4.

    int index = (nodeKey - 1) \* 4 + attributeIndex;

    // Setze den neuen Wert für das Attribut mit atomarer Operation.

    emscripten\_atomic\_store\_u32(

        (uint32\_t \*)globalNodesData + index,

        (uint32\_t)value);

}

Quellcode 9: Funktion setNodeAttribute

Es wurde jedoch kein signifikanter Unterschied in Bezug auf Performance oder Speicherverbrauch zwischen den beiden Methoden zur thread-sicheren Datenmanipulation festgestellt, weshalb beide Implementierungen als gleichwertig effizient angesehen werden.

### wasm\_actor\_webWorkers

Der Ansatz, den dSatur-Algorithmus im Aktoren Modell zu implementieren, war weniger komplex, da WebWorker in separaten Kontexten laufen und somit voneinander isolierte Speicherbereiche haben. Allerdings ist eine Wrapper-Klasse notwendig, um die Wasm-Module in den einzelnen Threads zu laden. Die Wrapper-Klasse DSaturWasmModuleWrapper.js ist in diesem Fall einfacher gehalten, da immer nur das gleiche Wasm-Modul geladen wird und kein geteilter Speicher erforderlich ist.

Das Skript test.js ist jedoch komplexer, da es als Aktor fungiert und daher die Funktionen updateSaturation, selectNode, lockAdjacentNodes und unlockAdjacentNodes implementieren muss. Die Datenmanipulation muss jedoch nicht thread-sicher erfolgen, da die Funktionen im Hauptthread sequenziell ausgeführt werden. Außerdem steuert das Skript die Aufgabenverteilung und führt die Ergebnisse aus den verschiedenen Threads korrekt zurück.

Im Folgenden möchten wir die wesentlichen Komponenten des Aktors näher erläutern:

Aktor-Komponenten:

* workers: Ein Array, das die WebWorker-Instanzen enthält, die die Knotenfärbung durchführen.
* nodeQueue: Eine Warteschlange, in der Knoten gespeichert werden, die gefärbt werden sollen, sobald ein Worker verfügbar ist.
* activeWorkers: Ein Set zur Nachverfolgung, welche Worker gerade Aufgaben ausführen.

Funktionale Komponenten:

* initializeWorkerPool: Diese Funktion erstellt und initialisiert die gewünschte Anzahl an WebWorkern, basierend auf der Thread-Anzahl. Die Worker werden im workers-Array gespeichert, und Nachrichten von den Workern werden in der Funktion processWorkerResult verarbeitet.
* processWorkerResult: Diese Funktion nimmt die gefärbten Knoten vom WebWorker entgegen, setzt die Farbe des Knotens im Speicherbereich des Aktors, aktualisiert die Farbe im Diagramm, führt die Funktionen updateSaturation und unlockAdjacentNode aus und entfernt den Worker aus activeWorkers. Danach wird der Worker zurück zum workers-Array hinzugefügt. Die Funktion prüft, ob alle Knoten gefärbt sind. Wenn nicht, wird addColorableNodeToQueue aufgerufen, um einen neuen Knoten der nodeQueue hinzuzufügen, gefolgt von processNextTask, um den nächsten Knoten einem verfügbaren WebWorker zur Bearbeitung zuzuweisen. Sind alle Knoten gefärbt, wird ein Promise zurückgegeben, das den Test beendet, und alle Worker werden terminiert.
* processNextTask: Überprüft, ob die nodeQueue gefüllt ist und ein Worker zur Verfügung steht. Entfernt in diesem Fall einen Knoten aus der Warteschlange und weist ihn einem Worker über sendNodeToWorker zu.
* addColorableNodeToQueue: Ruft selectNode auf, um den nächsten Knoten zur Färbung auszuwählen. Sperrt die angrenzenden Knoten mittels lockAdjacentNodes und übergibt den Knoten an sendNodeToWorker.
* sendNodeToWorker: Entnimmt einen verfügbaren Worker aus dem workers-Array und fügt ihn zum activeWorkers-Set hinzu. Übermittelt dem Worker mittels postMessage Informationen über den zu färbenden Knoten, die vollständige Knotenliste, die Kantenliste und die Anzahl der Terme. Ist kein Worker verfügbar, wird der Knoten zur nodeQueue hinzugefügt.

Die Funktion startTest des test.js-Skripts initialisiert lediglich das Promise, auf dessen Abschluss gewartet wird, um den Test zu beenden. Anschließend werden die Funktionen initializeWorkerPool und startActorRoutine sequenziell ausgeführt. Die Funktion startActorRoutine ruft einmalig die Funktion addColorableNodeToQueue für jeden Thread auf. Dadurch wird gewährleistet, dass alle Knoten durch die WebWorker verarbeitet werden, da diese nach der Färbung eines Knotens die Funktion processWorkerResult aufrufen, welche ihnen wiederum neue Knoten zur Färbung zuweist.

Die WebWorker, die das Skript colorNodeDSaturWorker.js ausführen, sind wie folgt strukturiert:

Der Worker wartet auf eine Nachricht des Aktors. Sobald eine solche Nachricht übermittelt wird, entnimmt er die Informationen über den zu färbenden Knoten (node), die Kanten (links), die Knoten (nodes) sowie die Anzahl der Terme (terms). Diese Daten werden bei jeder Aufgabe vollständig in den Kontext des WebWorkers geklont. Anschließend wird das Wasm-Modul mithilfe des DSaturWasmModuleWrapper initialisiert. Dieser Schritt erfolgt nur beim ersten Aufruf des WebWorkers, danach steht das Wasm-Modul kontinuierlich zur Verfügung.

Die Datenübertragung erfolgt über einen ArrayBuffer und die vom Wasm-Modul bereitgestellte malloc-Funktion, um die Daten in den Heap des Wasm-Moduls zu schreiben. Ein ArrayBuffer ist ausreichend, da die Daten nicht zwischen Workern geteilt werden. Schließlich wird mithilfe der Zeiger auf die Speicherbereiche die processNode-Funktion des Wasm-Moduls aufgerufen.

Um unseren C++-Code, der den dSatur-Algorithmus für einen einzelnen Knoten ausführt, mithilfe von Emscripten in WebAssembly zu kompilieren, wurde folgender Befehl (Quellcode 7: emscripten Befehl für wasm\_actor \_webWorkers) verwendet:

emcc dSaturSingleNode.cpp -o

dSaturSingleNode.js -s

EXPORTED\_FUNCTIONS='["\_malloc","\_free","\_processNode"]' -s

Quellcode 10: emscripten Befehl für wasm\_actor\_webWorkers

Die processNode-Funktion des Wasm-Moduls deserialisiert die Daten in std::vector-Structs vom Typ node und link, die ähnlich aufgebaut sind wie in den vorherigen Implementierungen, jedoch mit dem Unterschied, dass die Attribute des node-Struct als einfache int-Werte definiert sind. Anschließend wird die Funktion dSaturSingleNode ausgeführt, welche die Farbe für einen einzelnen Knoten berechnet. Danach wird die calculatePiLeibniz Funktion aufgerufen. Schließlich gibt die Funktion das Ergebnis als Farbwert zurück.

### wasm\_actor\_pthreads

Der Implementierungsansatz wasm\_actor\_pthreads ähnelt in seiner JavaScript-Struktur dem Ansatz wasm\_sharedMemory\_pthreads. Die Wasm-Modul-Seite gestaltete sich jedoch als komplex in der Implementierung, da keine isolierten Speicherbereiche zwischen den Threads und dem Hauptthread existieren. Es gibt in C++ kein native Nachrichtenverarbeitungssystem wie postMessage in JavaScript zur Koordination der Threads. Dennoch wurde versucht, so nah wie möglich an der Implementierung von wasm\_actor\_webWorkers zu bleiben, um einen besseren Vergleich zu ermöglichen.

Bis zur Verarbeitung im Wasm-Modul ist das Vorgehen auf JavaScript-Seite mit dem Ansatz wasm\_sharedMemory\_pthreads identisch. Da die Funktion im JavaScript-Hauptkontext aufgerufen wird, ist kein zusätzliches Laden des Wasm-Moduls erforderlich. Die startTest-Funktion bleibt ebenfalls unverändert. Außerdem bleibt auch der Befehl zum Kompilieren des C++-Codes zu WebAssembly unverändert.

Die processGraph-Funktion deserialisiert die Daten, ähnlich wie in den bisherigen Ansätzen, in std::vector-Strukturen der Klassen Node und Link, wobei in diesem Fall auch einfache Integer-Attribute verwendet werden. Danach wird ein Objekt der Actor-Klasse mit der übergebenen Anzahl an Threads und den Graph-Daten initialisiert. Schließlich wird die Funktion emscripten\_set\_main\_loop aufgerufen, um das Wasm-Modul anzuweisen, eine Funktion kontinuierlich auszuführen. In diesem Fall nutzen wir dies, um die executeMainThreadActions-Funktion des erstellten Actor Objekts kontinuierlich auszuführen.

Darüber hinaus wird eine Klasse TaskQueue implementiert, die eine threadsichere Warteschlange zur Verwaltung von Aufgaben bereitstellt, welche von mehreren Threads parallel verarbeitet werden. Mithilfe von Mutex und Bedingungsvariablen wird ein koordinierter Zugriff ermöglicht, sodass Threads effizient Aufgaben aus der Warteschlange entnehmen können, ohne Konflikte zu verursachen oder unnötige Ressourcen zu beanspruchen. Die Funktionsweise ist wie folgt beschrieben:

**Konstruktor:**

* TaskQueue(): Der Konstruktor initialisiert die shutdown-Variable mit false, was bedeutet, dass die Warteschlange aktiv ist.

**Öffentliche Methoden:**

* enqueue:
  + Beschreibung: Fügt eine neue Aufgabe (Task) in die Warteschlange ein.
  + Implementierung:
    - Sperrt den kritischen Bereich mit einem std::lock\_guard<std::mutex>.
    - Fügt die Aufgabe zur tasks-Warteschlange hinzu.
    - Signalisiert durch notify\_one der Bedingungsvariablen cond, dass eine neue Aufgabe verfügbar ist.
* dequeue:
  + Beschreibung: Entfernt die erste verfügbare Aufgabe aus der Warteschlange und gibt sie zurück.
  + Rückgabewert: Die nächste verfügbare Aufgabe als std::function<void()> oder nullptr, wenn die Warteschlange leer ist.
  + Implementierung:
    - Sperrt den kritischen Bereich mit einem std::unique\_lock<std::mutex>.
    - Wartet auf die Bedingung (shutdown oder tasks ist nicht leer), mit cond.wait.
    - Gibt nullptr zurück, wenn tasks leer ist.
    - Entfernt die Aufgabe aus tasks und gibt sie zurück.
* close:
  + Beschreibung: Signalisiert der Warteschlange, dass sie geschlossen wird.
  + Implementierung:
    - Setzt shutdown auf true.
    - Weckt alle wartenden Threads mit notify\_all auf.
* isEmpty:
  + Beschreibung: Überprüft, ob die Warteschlange leer ist.
  + Rückgabewert: true, wenn die Warteschlange leer ist, sonst false.
  + Implementierung:
    - Sperrt den kritischen Bereich mit einem std::lock\_guard<std::mutex>.
    - Prüft den Status von tasks.

**Private Attribute:**

* std::mutex mutex: Ein Mutex zur Synchronisation des Zugriffs auf die tasks-Warteschlange.
* std::condition\_variable cond: Eine Bedingungsvariable, die verwendet wird, um Threads zu benachrichtigen, dass Aufgaben verfügbar sind oder die Warteschlange geschlossen wurde.
* std::queue<std::function<void()>> tasks: Eine Warteschlange von Aufgaben, die durch die Threads abgearbeitet werden.
* std::atomic<bool> shutdown: Eine atomare Variable, die signalisiert, ob die Warteschlange geschlossen ist.

Die Actor-Klasse implementiert ein Aktorenmodell in C++, welche die Aufgabenverteilung zwischen TaskQueue und workerThreads organisiert. Es gibt zwei zentrale Warteschlangen:

Task Queue: Weist Aufgaben den P-Threads zu, indem sie Aufgaben nacheinander aus der Warteschlange entnimmt und an die Threads verteilt.

Main Thread Action Queue: Die Hauptthread-Warteschlange, die im zentralen Hauptthread läuft, nimmt die Ergebnisse der Threads entgegen und fügt neue Aufgaben in die TaskQueue ein.

Diese Struktur ermöglicht es, dass alle Aktionen, welche die zentralen Graph-Daten verändern, nur im Hauptthread durchgeführt werden. Dies gewährleistet einen threadsicheren Zugriff auf die Daten und sorgt so für eine konsistente und parallele Ausführung der Aufgaben über mehrere Threads hinweg. Die Funktionsweise der Actor-Klasse ist wie folgt beschrieben:

**Konstruktor Actor:**

* Parameter: numThreads legt die Anzahl der zu startenden Threads fest. nodes und links repräsentieren den Graphen. terms wird für die Pi-Berechnung verwendet.
* Implementierung:
* Initialisiert ein std::promise-Objekt (tasksCompletedPromise) zur Aufgabenverfolgung und ordnet Attribute für Knoten (nodes) und Kanten (links) zu.
* Erzeugt workerThreads (Worker-Threads) als std::thread-Objekte. Diese Threads entnehmen Aufgaben aus der TaskQueue und führen sie aus, solange nicht alle Knoten gefärbt sind.
* Fügt initial Aufgaben hinzu, indem die Methode addNextTask für jeden Thread aufgerufen wird.

**Destruktor ~Actor:**

* Beschreibung: Schließt die TaskQueue, wartet darauf, dass alle Threads die Verarbeitung beenden (join), löscht das tasksCompletedPromise-Objekt und führt die Hauptthread-Aktionen (executeMainThreadActions) aus.

**Öffentliche Methoden:**

* waitForTasksCompletion:
* Beschreibung: Wartet darauf, dass alle Aufgaben abgeschlossen sind, und führt danach die Aktionen im Hauptthread aus.
* enqueueMainThreadAction:
* Beschreibung: Fügt eine Aktion zur Warteschlange des Hauptthreads hinzu, um threadübergreifende Aktionen zu planen.
* Implementierung: Sichert den Zugriff auf die Warteschlange durch ein Mutex (std::lock\_guard) und fügt die Aktion in die mainThreadActionsQueue ein.
* executeMainThreadActions:
* Beschreibung: Führt alle Aktionen im Hauptthread aus.
* Implementierung:
* Entnimmt und führt Aktionen aus der mainThreadActionsQueue aus.
* Falls alle Knoten gefärbt sind, wird die Schleife durch emscripten\_cancel\_main\_loop gestoppt, und threadsFinishedJS wird aufgerufen.

**Private Methoden und Attribute:**

* addNextTask:
* Beschreibung: Fügt die nächste Färbeaufgabe zur TaskQueue hinzu.
* Implementierung:
* Wählt den nächsten Knoten zur Färbung durch selectNode.
* Sperrt angrenzende Knoten (lockAdjacentNodes).
* Fügt eine neue Aufgabe in die TaskQueue ein, um den Knoten zu färben (dSaturSingleNode), die Hauptthread-Aktionen zu planen und die Pi-Berechnung durchzuführen.
* lockAdjacentNodes und unlockAdjacentNodes:
* Beschreibung: Sperren bzw. entsperren angrenzende Knoten, um parallele Zugriffe zu vermeiden.
* updateSaturation:
* Beschreibung: Aktualisiert den Sättigungsgrad angrenzender Knoten.
* countLinksForNode:
* Beschreibung: Zählt die Anzahl der Kanten eines Knotens.
* selectNode:
* Beschreibung: Wählt den nächsten Knoten zur Färbung basierend auf dem Sättigungsgrad (Weight) und der Anzahl der Kanten aus.
* findNodeByKey:
* Beschreibung: Sucht den Knoten durch seine ID (key).
* calculatePiLeibniz:
* Beschreibung: Berechnet Pi mithilfe der alternierenden Leibniz-Reihe.
* dSaturSingleNode:
* Beschreibung: Berechnet die Farbe eines Knotens anhand des DSatur-Algorithmus und führt die Pi-Berechnung durch.
* areAllNodesColored:
* Beschreibung: Überprüft, ob alle Knoten gefärbt sind.

# Ergebnisse

# Diskussion

# Ausblick

# Anhang A Abbildungen

# Anhang B Quellcode:

/\*\*

 \* Implementiert den dSatur-Algorithmus als Thread-Funktion.

 \* Farbkodierung für Graphknoten durch rekursive Auswahl

 \* des optimalen Knotens und Zuweisung einer konfliktfreien Farbe.

 \*

 \* @param arg Zeiger auf ThreadData mit Knoten, Links und Daten.

 \* @return nullptr nach Ausführungsende.

 \*

 \* Vermeidet Race-Conditions und Deadlocks. Nutzt die

 \* Pi-Berechnung zur Simulation zusätzlicher Thread-Arbeit.

 \*/

void \*dSatur(void \*arg)

{

    auto data = static\_cast<ThreadData \*>(arg);

    auto &nodes = \*(data->nodes);

    auto &links = \*(data->links);

    int terms = data->terms;

    int threadId = data->threadId;

    Node \*node = selectNode(nodes, links);

    while (node != nullptr)

    {

        int color = 1;

        bool adjacentColored;

        do

        {

            adjacentColored = false;

            for (const auto &link : links)

            {

                if (link.from == node->key || link.to == node->key)

                {

                    auto &targetNode = (link.from == node->key) ?

                        \*std::find\_if(nodes.begin(), nodes.end(),

                            [&](const Node &n)

                             { return n.key == link.to; }) :

                        \*std::find\_if(nodes.begin(), nodes.end(),

                            [&](const Node &n)

                             { return n.key == link.from; });

                    if (targetNode.color.load() == color)

                    {

                        adjacentColored = true;

                        break;

                    }

                }

            }

            if (adjacentColored) ++color;

        } while (adjacentColored);

        if (node->color.load() == 0 && node->lock.load() == 0)

        {

            lockAdjacentNodes(\*node, nodes, links);

            updateSaturation(node->key, color, nodes, links);

            node->color.store(color);

            calculatePiLeibniz(terms + color);

            unlockAdjacentNodes(\*node, nodes, links);

            emscripten\_async\_run\_in\_main\_runtime\_thread(

                EM\_FUNC\_SIG\_VIII, changeNodeColorCallback,

                node->key, color, threadId);

        }

        node = selectNode(nodes, links);

    }

    return nullptr;

}

Quellcode-Anhang 1: Funktion dSatur

/\*\*

 \* Aktualisiert die Sättigung angrenzender Knoten, wenn sie noch

 \* nicht gefärbt sind.

 \*

 \* @param nodeKey Schlüssel des aktuellen Knotens.

 \* @param color Farbe des aktuellen Knotens.

 \* @param nodes Referenz auf die Liste aller Knoten.

 \* @param links Konstante Referenz auf die Liste aller Verbindungen.

 \*/

void updateSaturation(int nodeKey, int color,

                      std::vector<Node> &nodes,

                      const std::vector<Link> &links)

{

    for (const auto &link : links)

    {

        Node \*targetNode = nullptr;

        if (link.from == nodeKey)

        {

            targetNode = findNodeByKey(nodes, link.to);

        }

        else if (link.to == nodeKey)

        {

            targetNode = findNodeByKey(nodes, link.from);

        }

        if (targetNode != nullptr && targetNode->color.load() == 0)

        {

            targetNode->saturation.fetch\_add(1);

        }

    }

}

Quellcode-Anhang 2: Funktion updateSaturation

/\*\*

 \* Sperrt den aktuellen Knoten und alle angrenzenden Knoten.

 \*

 \* @param node Referenz auf den aktuellen Knoten.

 \* @param nodes Referenz auf die Liste aller Knoten.

 \* @param links Konstante Referenz auf die Liste aller Verbindungen.

 \*/

void lockAdjacentNodes(Node &node, std::vector<Node> &nodes,

                       const std::vector<Link> &links)

{

    node.lock.fetch\_add(1);

    for (const auto &link : links)

    {

        if (link.from == node.key || link.to == node.key)

        {

            int targetNodeKey =

             (link.from == node.key) ? link.to : link.from;

            auto targetNodeIter = std::find\_if(

                nodes.begin(), nodes.end(),

                [targetNodeKey](const Node &n)

                 { return n.key == targetNodeKey; });

            if (targetNodeIter != nodes.end())

            {

                targetNodeIter->lock.fetch\_add(1);

            }

        }

    }

}

Quellcode-Anhang 3: Funktion lockAdjacentNodes

/\*\*

 \* Entsperren des aktuellen Knotens und aller angrenzenden Knoten.

 \*

 \* @param node Referenz auf den aktuellen Knoten.

 \* @param nodes Referenz auf die Liste aller Knoten.

 \* @param links Konstante Referenz auf die Liste aller Verbindungen.

 \*/

void unlockAdjacentNodes(Node &node, std::vector<Node> &nodes,

                         const std::vector<Link> &links)

{

    node.lock.fetch\_sub(1); // Entsperren des aktuellen Knotens

    for (const auto &link : links)

    {

        if (link.from == node.key || link.to == node.key)

        {

            int targetNodeKey =

             (link.from == node.key) ? link.to : link.from;

            auto targetNodeIter = std::find\_if(

                nodes.begin(), nodes.end(),

                [targetNodeKey](const Node &n)

                 { return n.key == targetNodeKey; });

            if (targetNodeIter != nodes.end())

            {

                targetNodeIter->lock.fetch\_sub(1);

            }

        }

    }

}

Quellcode-Anhang 4: Funktion unlockAdjacentNodes

/\*\*

 \* Berechnet den Gini-Koeffizienten für eine Verteilung von Aufgaben

 \* über Threads, um die Ungleichheit in der Verteilung zu messen.

 \* Der Gini-Koeffizient ist ein Maß für die Ungleichheit, wobei 0

 \* perfekte Gleichheit (jeder Wert ist gleich) und 1 maximale

 \* Ungleichheit (alle Werte sind ungleich) bedeutet.

 \*

 \* @returns {number} Der berechnete Gini-Koeffizient.

 \*

 \* Die Funktion sortiert zuerst die Aufgaben, berechnet dann die

 \* kumulative Summe der Aufgaben und nutzt diese Werte, um den

 \* Gini-Koeffizienten zu ermitteln.

 \*/

function calculateGiniCoefficient() {

    // Sortieren der Aufgabenanzahl je Thread

    const sortedTasks = threadNodeCount.slice().sort((a, b) => a - b);

    const n = sortedTasks.length; // Anzahl der Threads

    let kumulativeSummen = 0;

    let summeDerAufgaben = 0;

    // Berechnen der kumulativen Summen und der Gesamtsumme der Aufgaben

    for (let i = 0; i < n; i++) {

      kumulativeSummen += sortedTasks[i] \* (i + 1);

      summeDerAufgaben += sortedTasks[i];

    }

    // Berechnen des Gini-Koeffizienten,  
 // wenn die Summe der Aufgaben nicht 0 ist

    const gini = summeDerAufgaben

      ? (2 \* kumulativeSummen) / (n \* summeDerAufgaben) - (n + 1) / n

      : 0;

    return gini; // Rückgabe des berechneten Gini-Koeffizienten

  }

Quellcode-Anhang 5: Funktion calculateGiniCoefficient

/\*\*

 \* Serialisiert die Knoten- und Verbindungsdaten

 \* in den `int32View`-Puffer,

 \* der auf einen SharedArrayBuffer verweist.

 \* Die Knoten und Kanten

 \* werden in einem kompakten Format gespeichert,

 \* sodass das WebAssembly-Modul

 \* sie korrekt verarbeiten kann.

 \*/

function serializeGraph() {

  // Serialisiere die Knoteninformationen in den Puffer

  nodes.forEach((node, index) => {

      // Bestimme die Startposition für jeden Knoten

      // (4 Int32-Werte pro Knoten)

      let baseIndex = index \* 4;

      int32View[baseIndex] = node.key; // Knoten-ID

      int32View[baseIndex + 1] = node.color; // Aktuelle Knotenfarbe

      int32View[baseIndex + 2] = node.weight; // Gewicht des Knotens

      int32View[baseIndex + 3] = node.lock; // Sperrstatus des Knotens

  });

  // Serialisiere die Verbindungen in den Puffer

  links.forEach((link, index) => {

      // Bestimme die Startposition für jede Kante

      // Beginnt nach den Knoteninfos und hat 2 Int32-Werte pro Kante

      let baseIndex = nodes.length \* 4 + index \* 2;

      int32View[baseIndex] = link.from; // Ursprungknoten der Kante

      int32View[baseIndex + 1] = link.to; // Zielknoten der Kante

  });

}

Quellcode-Anhang 6: Funktion serializeGraph

/\*\*

 \* Verarbeitet den Graphen,

 \* indem die Rohdaten deserialisiert und die Threads zur

 \* Ausführung des dSatur-Algorithmus gestartet werden.

 \*

 \* @param nodesData Zeiger auf den Speicherbereich

 \*        mit den Knoteninformationen.

 \* @param nodeCount Anzahl der Knoten.

 \* @param linksData Zeiger auf den Speicherbereich

          mit den Verbindungsinformationen.

 \* @param linkCount Anzahl der Verbindungen.

 \* @param terms Anzahl der Terme für die Berechnung der Leibniz-Reihe.

 \* @param numThreads Anzahl der Threads, die gestartet werden sollen.

 \*/

void processGraph(int32\_t \*nodesData,

 int nodeCount,

 int32\_t \*linksData,

 int linkCount,

 int terms,

 int numThreads) {

    // Erstellen eines Vektors für Knoten und Initialisierung

    std::vector<Node> nodes;

    std::vector<Link> links(linkCount);

    // Konvertiere die Rohdaten in eine Liste von `Node`-Strukturen

    for (int i = 0; i < nodeCount; ++i) {

        nodes.emplace\_back(nodesData[i \* 4],

                           nodesData[i \* 4 + 1],

                           nodesData[i \* 4 + 2],

                           nodesData[i \* 4 + 3]);

    }

    // Konvertiere die Rohdaten in eine Liste von `Link`-Strukturen

    for (int i = 0; i < linkCount; ++i) {

        links[i] = {linksData[i \* 2],

                    linksData[i \* 2 + 1]};

    }

    // Starte die Threads zur Ausführung des dSatur-Algorithmus

    invokeThreads(numThreads, nodes, links, terms);

}

Quellcode-Anhang 7: Funktion processGraph

/\*\*

 \* Startet die spezifizierte Anzahl an Threads

 \* und führt den dSatur-Algorithmus

 \* auf den übergebenen Knoten- und Verbindungsdaten aus.

 \*

 \* @param numThreads Anzahl der zu startenden Threads.

 \* @param nodes Referenz auf den Vektor der Knoten.

 \* @param links Referenz auf den Vektor der Verbindungen.

 \* @param terms Anzahl der Terme für

 \*  die Berechnung der Leibniz-Reihe.

 \*/

void invokeThreads(

     int numThreads,

     std::vector<Node> &nodes,

     std::vector<Link> &links,

     int terms) {

    // Array zur Speicherung der Thread-IDs

    pthread\_t threads[numThreads];

    // Vektor zur Speicherung der Zeiger auf ThreadData-Objekte

    std::vector<ThreadData \*> threadData;

    // Erstelle und starte die spezifizierte Anzahl an Threads

    for (int i = 0; i < numThreads; ++i) {

        ThreadData \*data = new ThreadData(&nodes, &links, terms, i);

        // Füge den Zeiger auf das ThreadData-Objekt zum Vektor hinzu

        threadData.push\_back(data);

        // Erstelle einen neuen Pthread und starte den dSatur-Algorithmus

        pthread\_create(&threads[i], nullptr, dSatur, data);

    }

    // Warte auf die Beendigung aller Threads

    for (int i = 0; i < numThreads; ++i) {

        pthread\_join(threads[i], nullptr);

    }

    // Speicher für jedes ThreadData-Objekt freigeben

    for (ThreadData \*data : threadData) {

        delete data;

    }

    // Signalisiert dem JavaScript-Code, dass alle Threads fertig sind

    threadsFinishedJS();

}

Quellcode-Anhang 8: Funktion invokeThreads

# Literaturverzeichnis

An, D., & Meenan, P. (07 2016). *Why Marketers Should Care About Mobile Page Speed.* think with Google. Abgerufen am 05. 02 2024 von https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-load-time/

Bera, R. (22. 12 2022). *JavaScript Execution Context – How JS Works Behind the Scenes*. (freeCodeCamp) Abgerufen am 17. 02 2024 von https://www.freecodecamp.org/news/how-javascript-works-behind-the-scene-javascript-execution-context/

Berners-Lee, T. (1990). *Information Management: A Proposal.* CERN. Abgerufen am 05. 02 2024 von https://cds.cern.ch/record/369245/files/dd-89-001.pdf

Bewersdorff, J. (2014). Multithreading mit JavaScript. In *Objektorientierte Programmierung mit JavaScript* (S. 310-315). Springer.

Borchers, D. (10. 11 2003). *10 Jahre Mosaic*. (heise) Abgerufen am 15. 02 2024 von https://www.heise.de/news/10-Jahre-Mosaic-88313.html

Chromium Blog. (20. 06 2019). *WebAssembly brings Google Earth to more browsers*. (Google) Abgerufen am 22. 02 2024 von https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html

Clark, L. (28. 02 2017). *What makes WebAssembly fast?* (hacks.mozilla) Abgerufen am 22. 02 2024 von https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/

Corusa, A., Predel, J., & Schöne, N. (2020). *Eine Marktübersicht der Blockchain in der Energiewirtschaft.* Berlin: Technische Universität Berlin. Von https://depositonce.tu-berlin.de/bitstream/11303/11657.2/9/blockchain\_energiewirtschaft\_2020\_v2.pdf abgerufen

CRYREX Enterprise. (2023). *Deep Dive Into Javascript Engines — Blazingly Fast*. Abgerufen am 16. 02 2024 von https://cyrexenterprise.com/deep-dive-into-javascript-engines-blazingly-fast-%E2%9A%A1%EF%B8%8F/

emscripten. (2015). *About Emscripten*. Abgerufen am 26. 02 2024 von https://emscripten.org/docs/introducing\_emscripten/about\_emscripten.html

emscripten. (kein Datum). *Pthreads support*. Abgerufen am 25. 02 2024 von https://emscripten.org/docs/porting/pthreads.html

FFmpeg. (kein Datum). *About FFmpeg*. Abgerufen am 03. 03 2024 von https://ffmpeg.org/about.html

FFmpeg. (kein Datum). *ffmpeg.wasm Overview*. Abgerufen am 03. 03 2024 von https://ffmpegwasm.netlify.app/docs/overview/

FFmpeg. (kein Datum). *ffmpeg.wasm Performance*. Abgerufen am 03. 03 2024 von https://ffmpegwasm.netlify.app/docs/performance

Flanagan, D. (2020). *JavaScript: The Definitive Guide.* O'Reilly Media.

freeCodeCamp. (01. 02 2020). *Just in Time Compilation Explained*. Abgerufen am 16. 02 2024 von https://www.freecodecamp.org/news/just-in-time-compilation-explained/

Gallant, G. (2019). Meet WebAssembly. In *WebAssembly in Action* (S. 3-17). Shelter Island: Manning Publications Co.

Gallant, G. (2019). Threading: Web workers an pthreads. In *WebAssembly in Action* (S. 195-218). Shelter Island: Manning.

Gallant, G. (2019). *WebAssembly in Action.* Shelter Island: Manning.

geeks for geeks. (03. 01 2024). *Thread Pool in C++*. Abgerufen am 29. 02 2024 von https://www.geeksforgeeks.org/thread-pool-in-cpp/

Goldberg, I., & Wagner, D. (01. 01 1996). *Randomness and the Netscape Browser.* Abgerufen am 15. 02 2024 von https://www.hit.bme.hu/~buttyan/courses/BMEVIHIM132/abib/03-prng/Goldberg96.pdf

Hunter, T., & English, B. (2021). *Multithreaded JavaScript.* Boston: O'Reilly.

Hunter, T., & English, B. (2021). Webassembly. In *Multithreaded JavaScript* (S. 155 - 167). Boston: O'Reilly.

Hunter, T., & English, B. (2021). What Are Threads? In *Multithreaded JavaScript Concurrency Beyond the Event Loop* (S. 3). Sebastopol: O'Reilly.

Ippolito, G. (2004). *POSIX thread (pthread) libraries*. Abgerufen am 25. 02 2024 von https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

java T point. (kein Datum). *What is Java Bytecode?* Abgerufen am 26. 02 2024 von https://www.javatpoint.com/java-bytecode

Kamal, R. (05. 07 2023). *Understanding JavaScript's Single-Threaded Nature*. (Dev.to) Abgerufen am 17. 02 2024 von https://dev.to/roktim32/understanding-javascripts-single-threaded-nature-5cd6

Kamani, S. (14. 03 2016). *How Is Javascript Asynchronous And Single Threaded?* Abgerufen am 17. 02 2024 von https://www.sohamkamani.com/javascript/single-thread-and-async/

Kavi, K. (9 1998). *Multithreading Implementations.* Abgerufen am 25. 02 2024 von https://www.researchgate.net/publication/2734966\_Multithreading\_Implementations

Kitamura, E., & Ubl, M. (20. 10 2012). *Web Worker-Grundlagen*. (web.dev) Abgerufen am 25. 02 2024 von https://web.dev/articles/workers-basics?hl=de#communicating\_with\_a\_worker\_via\_message\_passing

Kolosovskyi, I. (08. 10 2016). *Is JavaScript Single-Threaded*. (redgate) Abgerufen am 17. 02 2024 von https://www.red-gate.com/simple-talk/development/javascript/javascript-single-threaded/

LLVM. (kein Datum). *The LLVM Compiler Infrastructure*. Abgerufen am 11. 03 2024 von https://llvm.org/

mdn web docs. (kein Datum). *Memory management*. Abgerufen am 19. 02 2024 von https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory\_management

mdn web docs. (kein Datum). *Using Web Workers*. Abgerufen am 25. 02 2024 von https://developer.mozilla.org/en-US/docs/Web/API/Web\_Workers\_API/Using\_web\_workers

mdn web docs. (kein Datum). *WebAssembly Concepts*. Abgerufen am 22. 02 2024 von https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

mdn web docs. (kein Datum). *WebAssembly Concepts*. Abgerufen am 2024. 02 22 von https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

Milner, J. (22. 07 2019). *Using WebAssembly with Web Workers*. (sitepen) Abgerufen am 08. 02 2024 von https://www.sitepen.com/blog/using-webassembly-with-web-workers

Nadiger, A. (19. 06 2023). *Web-Assembly(Wasm) introduction*. Abgerufen am 22. 02 2024 von https://www.linkedin.com/pulse/web-assemblywasm-introduction-amit-nadiger

Onwuemene, J. (kein Datum). *How to Handle Memory Management in JavaScript*. (turing.com) Abgerufen am 19. 02 2024 von https://www.turing.com/kb/handling-memory-management-in-javascript

OpenCSF. (kein Datum). *Race Conditions and Critical Sections*. Abgerufen am 25. 02 2024 von https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/RaceConditions.html

Padmanabhan, S., & Jha, P. (22. 04 2019). *WebAssembly at eBay: A Real-World Use Case*. (Ebay) Abgerufen am 04. 03 2024 von https://innovation.ebayinc.com/tech/engineering/webassembly-at-ebay-a-real-world-use-case/

Pitchkites, M. (13. 01 2024). *The Fastest Browser Options in 2024: What’s the Best Web Browser for Speed?* (Cloudwards) Abgerufen am 16. 02 2024 von https://www.cloudwards.net/fastest-browser/

Rauschmayer, A. (2014). *Speaking JavaScript.* O'Reilly Media.

Rosenwasser, D. (09. 03 2022). *A Proposal For Type Syntax in JavaScript*. (Microsoft) Abgerufen am 26. 02 2024 von https://devblogs.microsoft.com/typescript/a-proposal-for-type-syntax-in-javascript/

Sagore, R. (02. 01 2019). *Stack, Event & Job Queue in Javascript*. (Medium) Abgerufen am 18. 02 2024 von https://medium.com/@Rahulx1/understanding-event-loop-call-stack-event-job-queue-in-javascript-63dcd2c71ecd

Schöning, U. (1997). Komplexitätstheorie. In *Theoretische Informatik - kurzgefasst* (S. 151 - 182). Heidelberg: Spektrum.

Serverance, C. (02 2012). *JavaScript:Designing a Language in 10 Days.* Abgerufen am 15. 02 2024 von https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6155645

Sonal, S. (2. 3 2020). *MultiThreading vs MultiProcessing vs MultiTasking vs MultiProgramming: Brief Overview*. (Medium) Abgerufen am 25. 02 2024 von https://medium.com/@shwetank3/multithreading-vs-multiprocessing-vs-multitasking-vs-multiprogramming-brief-overview-3e83a3e7fec6

Souders, S. (2007). *High Performance Web Sites: Essential Knowledge for Front-End Engineers.* O'Reilly Media.

Stepanyan, I. (kein Datum). *WebAssembly-Threads aus C, C++ und Rust verwenden*. (web.dev) Abgerufen am 08. 02 2024 von https://web.dev/articles/webassembly-threads?hl=de

Surma. (15. 07 2019). *Is postMessage slow?* Abgerufen am 29. 02 2024 von https://surma.dev/things/is-postmessage-slow/index.html

Surma. (30. 06 2021). *The State Of Web Workers In 2021*. Abgerufen am 29. 02 2024 von https://www.smashingmagazine.com/2021/06/web-workers-2021/

Szewczyk, R., Stonehouse, K., Barbalace, A., & Spink, T. (17. 02 2023). *Leaps and bounds: analysing WebAssembly’s performance with a focus on bounds checking .* Abgerufen am 10. 03 2024 von https://research-repository.st-andrews.ac.uk/bitstream/handle/10023/27004/iiswc22\_paper90.pdf;jsessionid=DB5253A5D0A147D6E6AFDA37238CCF81?sequence=1

Taivalsaari, A., & Mikkonen, T. (2011). The Web as an Application Platform: The Saga Continues. IEEEE. doi:10.1109/SEAA.2011.35

Vailshery, L. (19. 01 2024). *Most used programming languages among developers worldwide as of 2023*. (statista) Abgerufen am 26. 02 2024 von https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

Verma, A. (13. 04 2021). *The JavaScript Event Loop: Explained*. (Medium) Abgerufen am 18. 02 2024 von https://towardsdev.com/event-loop-in-javascript-672c07618dc9

Wallace, E. (08. 06 2017). *WebAssembly cut Figma's load time by 3x*. (figma) Abgerufen am 22. 02 2024 von https://www.figma.com/de/blog/webassembly-cut-figmas-load-time-by-3x/

WebAssembly. (23. 11 2019). *AutoCAD Web App*. Abgerufen am 22. 02 2024 von https://madewithwebassembly.com/showcase/autocad/

WebAssembly. (kein Datum). *Feature Extensions*. Abgerufen am 22. 02 2024 von https://webassembly.org/features/

Webassembly. (kein Datum). *Made with WebAssembly*. Abgerufen am 04. 03 2024 von https://madewithwebassembly.com/

Wilson, B. (26. 03 2018). *Using web workers for safe, concurrent JavaScript*. (LogRocket) Abgerufen am 29. 02 2024 von https://blog.logrocket.com/using-webworkers-for-safe-concurrent-javascript-3f33da4eb0b2/

Wingler, M. (5. 12 2018). *WebAssembly: die Zukunft der Webapps*. Abgerufen am 05. 02 2024 von https://blog.doubleslash.de/webassembly-die-zukunft-der-webapps

Woo, H. (31. 05 2016). *Javascript Engine & Performance Comparison (V8, Chakra, Chakra Core)*. (Red Hat Developer) Abgerufen am 16. 02 2024 von https://developers.redhat.com/blog/2016/05/31/javascript-engine-performance-comparison-v8-charkra-chakra-core-2#javascript\_engines

Yan, Y., Tu, T., Zhao, L., Zhou, Y., & Wang, W. (11 2021). *Understanding the performance of webassembly applications.* Abgerufen am 10. 03 2024 von https://dl.acm.org/doi/abs/10.1145/3487552.3487827

Yuan, A., & Dukhan, M. (02. 09 2020). *Supercharging the TensorFlow.js WebAssembly backend with SIMD and multi-threading*. (Google) Abgerufen am 01. 03 2024 von https://blog.tensorflow.org/2020/09/supercharging-tensorflowjs-webassembly.html

# Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Die vorgelegte Arbeit hat weder in der gegenwärtigen noch in einer anderen Fassung schon einem anderen Fachbereich der Hochschule Ruhr West oder einer anderen wissenschaftlichen Hochschule vorgelegen.

Ort, Datum Unterschrift

1. LLVM (Low Level Virtual Machine) ist ein modulares Compiler-Framework, das zur Entwicklung von Compilern für verschiedene Programmiersprachen genutzt wird, um Code in optimierten Maschinencode zu übersetzen (LLVM, kein Datum). [↑](#footnote-ref-1)