Fachhochschule Aachen, Campus Jülich

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Seminararbeit

Entwicklung eines automatisierten Tools zur Verifikation von Slurm Prozess Pinning auf Höchstleistungsrechnern

Fachbereich Medizintechnik und Technomathematik
im Studiengang Angewandte Mathematik und Informatik

Jülich, den 2. Januar 2025

Erstellt von: Carina Himmels (Matr.-Nr.: 3566808)

1. Prüfer: Prof. Dr. rer. nat. Volker Sander

2. Prüfer: Thomas Breuer

Firma: Forschungszentrum Jülich GmbH

Institut: Jülich Supercomputing Centre

JÜLICH
Forschungszentrum

Eidesstattliche Erklärung

Hiermit versichere ich, Carina Himmels, dass ich die Seminararbeit mit dem Thema

Entwicklung eines automatisierten Tools zur Verifikation von Slurm Prozess Pinning auf Höchstleistungsrechnern

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Jülich, den 2. Januar 2025

Unterschrift der Studentin/des Studenten

Abstract zur Seminararbeit

Entwicklung eines automatisierten Tools zur Verifikation von Slurm Prozess Pinning auf Höchstleistungsrechnern

Das Jülich Supercomputing Centre (JSC) stellt Expertise im Bereich des High-Performance Computing (HPC) sowie Rechenzeit auf modernen Höchstleistungsrechnern zur Verfügung. Ein solcher Höchstleistungsrechner besteht aus einer Vielzahl miteinander verbundener Rechenknoten, welche jeweils mehrere Prozessoren mit zahlreichen Kernen enthalten.

Auf diesen Systemen spielt das sogenannte Pinning, bei dem Prozesse an spezifische Kerne gebunden werden, eine zentrale Rolle in der Optimierung von Anwendungen. Durch geschickt gewähltes Pinning können beispielsweise zeitintensive Fernspeicherzugriffe reduziert werden, was die Leistung von Anwendungen erheblich verbessert.

Zur Unterstützung der Anwender bei der Auswahl einer geeigneten Pinning-Konfiguration stellt das JSC eine webbasierte Oberfläche, das Pinning-Webtool[1], zur Verfügung. Dieses simuliert verschiedenste Pinning-Konfigurationen auf den HPC-Systemen des JSC und ermöglicht dadurch ein besseres Verständnis des Pinnings.

Im Rahmen dieser Seminararbeit wurde eine CI/CD-Pipeline entwickelt, welche das simulierte Pinning des Webtools verifiziert, indem ein Vergleich zwischen dem simulierten Pinning im Pinning-Webtool und dem tatsächlichen Pinning auf den HPC-Systemen durchgeführt wird. Das Hauptziel dieser Pipeline ist die frühzeitige Identifizierung von Abweichungen zwischen der Simulation des Pinnings im Webtool und dem tatsächlichen Pinning auf den HPC-Systemen, damit Abweichungen möglichst schnell behoben werden können und das Webtool weiterhin eine zuverlässige Unterstützung für die Nutzer der HPC-Systeme darstellt.

[1] `https://apps.fz-juelich.de/jsc/llview/pinning/index.html`

Inhaltsverzeichnis

Abbildungsverzeichnis

Code-Verzeichnis

Acronyme und Fachbegriffe

API Application Programming Interface
CI Continuous Integration
CD Continuous Delivery/Deployment
Cluster Gruppe von vernetzten Computern
CPU Central Processing Unit (Prozessor)
CSS Cascading Style Sheets
Daemon Hintergrundprozess, der ohne Interaktion mit Benutzern Aufgaben erfüllt
DOM Document Object Model
GPU Graphics Processing Unit (Grafikprozessor)
HPC High-Performance Computing
HPC-System siehe Cluster
HTML HyperText Markup Language
JSC Jülich Supercomputing Centre
JUSUF Jülich Support for Fenix (HPC-System des JSC)
Knoten parallel arbeitender Rechner in einem HPC-System
Logische Kerne ausführbare Hardware-Threads innerhalb eines physischen Kerns
MPI Message Passing Interface
NUMA Non-Uniform Memory Access
OpenMP Open Multi-Processing
Physische Kerne Kerne eines Prozessores
Pinning Bindung eines Prozesses oder Threads an bestimmte CPU-Kerne
Plug-In Software-Erweiterung
Prozess eine in sich abgeschlossene Ausführungeinheit, die aus einem oder mehr Threads besteht
Prozess-Affinität siehe Pinning
SMT Simultaneous Multi-Threading
SVG Scalable Vector Graphics
Task siehe Prozess
Thread Teil eines Prozesses
XML eXtensible Markup Language
YAML YAML Ain't Markup Language

1. Einleitung

1.1. Motivation

High-Performance Computing, kurz HPC, hat in den letzten Jahren in einer Vielzahl von Anwendungsgebieten, nicht zuletzt in der Wissenschaft und Industrie, erheblich an Bedeutung gewonnen. Am Jülich Supercomputing Centre (JSC) des Forschungszentrums Jülich spielt HPC ebenfalls eine zentrale Rolle, da das JSC Wissenschaftlern den Zugang zu Höchstleistungsrechnern sowie Rechenzeit auf diesen Systemen bietet und sie bei deren Benutzung unterstützt. Die Leistung der auf diesen Rechnern ausgeführten Anwendungen hängt dabei maßgeblich von der optimalen Verteilung der Prozesse auf die vorhandenen Ressourcen ab. Um den Nutzern der Höchstleistungsrechner eine anschauliche Darstellung dieser Prozessverteilung, dem sogenannten Pinning, zu bieten, wurde das Pinning-Webtool entwickelt, welches das Pinning auf den Höchstleistungsrechnern des JSC simuliert. Angesichts der großen Bedeutung der optimalen Prozessverteilung ist es essenziell, dass diese Simulation stets mit dem tatsächlichen Pinning auf den Höchstleistungsrechnern übereinstimmt. Aus diesem Grund ist es wichtig, dass mögliche Unterschiede frühzeitig erkannt und behoben werden können. Hierzu sollen regelmäßige Vergleiche mittels CI/CD automatisiert werden.

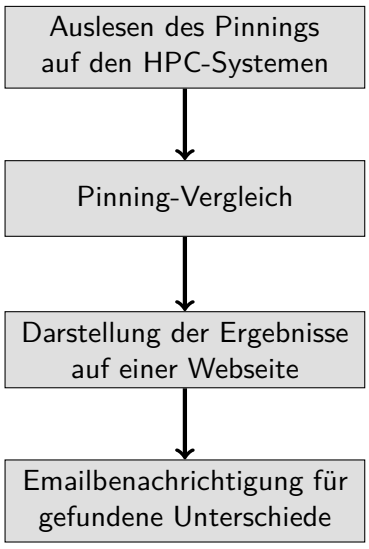

Abb. 1: Ablauf der zu implementierenden CI/CD-Pipeline

Genauer gesagt sollen die einzelnen Schritte, die in Abbildung 1 zu sehen sind, automatisiert durchgeführt werden. Es soll also zunächst das Pinning auf den HPC-Systemen ausgelesen und im Anschluss mit dem simulierten Pinning im Pinning-Webtool vergleichen werden, um mögliche Unterschiede zu erkennen. Anschließend sollen die Ergebnisse des Vergleichs möglichst übersichtlich auf einer Webseite dargestellt werden, damit die Ursachen der Unterschiede möglichst einfach analysiert werden können, und zum

Schluss sollen zusätzlich die Entwickler per Email informiert werden, falls Unterschiede entdeckt wurden, damit diese möglichst schnell behoben werden können. Insgesamt soll diese Automatisierung sicherstellen, dass das Pinning-Webtool auch langfristig eine sinnvolle und zuverlässige Unterstützung für Benutzer der HPC-Systeme des JSC darstellt. [1, 2]

1.2. High-Performance Computing

High-Performance Computing beschreibt den Einsatz mehrerer eng miteinander verbundener Computer, welche parallel als ein einziges System zusammenarbeiten. Durch diese Kombination von Rechenleistung können umfangreiche Aufgaben bewältigt werden, welche auf einem einzelnen handelsüblichen Computer nicht durchführbar wären, da der Speicherbedarf dieser Anwendungen zu groß oder der Anwendungsfall zu komplex wäre, um ihn in akzeptabler Zeit berechnen zu können. Dazu zählen beispielsweise komplexe Simulationen, Berechnungen und Datenanalysen. HPC-Systeme, sogenannte Cluster, zeichnen sich zusätzlich durch Hochleistungsnetzwerke und große Speicherkapazitäten aus, wodurch sie in der Lage sind, große Mengen an paralleler Verarbeitung durchzuführen. Sie sind dadurch sowohl in der Wissenschaft als auch in der Wirtschaft bei der Entdeckung neuer Erkenntnisse, der Optimierung von Produkten und der Verkürzung von Herstellungszeiten von großer Bedeutung. [3]

Vereinfacht ausgedrückt setzt sich ein Höchstleistungsrechner aus zahlreichen gekoppelten und parallel arbeitenden Rechnern zusammen, welche als Knoten bezeichnet werden. Jeder dieser Knoten wiederum enthält eine bestimmte Anzahl an Prozessoren (CPU) und zusätzlichen Grafik-Prozessoren (GPU). Weit verbreitet ist außerdem die Speicherarchitektur NUMA, kurz für Non-Uniform Memory Access, bei der jeder Knoten ein NUMA-System bildet. Ein solches System besteht aus einer Hauptplatine mit mehreren Sockets, die je eine CPU enthalten. Jeder dieser Sockets ist einem bestimmten Teil des Hauptspeichers zugeordnet, auf den sehr schnell zugegriffen werden kann. Auf Speicherbereiche, die einem Socket nicht zugeordnet worden sind, kann zwar auch zugegriffen werden, allerdings sind diese Fernspeicherzugriffe deutlich zeitaufwändiger, da eine spezielle Leitung verwendet werden muss. [1, 4]

Ein Beispiel für ein HPC-System, welches die NUMA-Speicherarchitektur verwendet, ist das Cluster JUSUF, welches vom JSC betrieben wird. Wie in Abbildung 2 zu sehen ist, besteht jeder Knoten in diesem System aus genau zwei Sockets mit je einem Prozessor. Jeder dieser Prozessoren enthält 64 physische Kerne, welche durch SMT, kurz für Simultaneous Multi-Threading, je zwei Hardware-Threads beinhalten, welche in der Abbildung durch je eine „0“ dargestellt werden. Insgesamt enthält jeder Prozessor somit 128 logische Kerne. Außerdem gibt es in jedem Socket vier NUMA-Domänen, also vier verschiedene lokale Speicherbereiche, mit je 16 physischen Kernen. Zur vereinfachten Darstellung werden einige Hardware-Threads in den NUMA-Domänen zwei bis sieben durch „...“ ersetzt.[5]

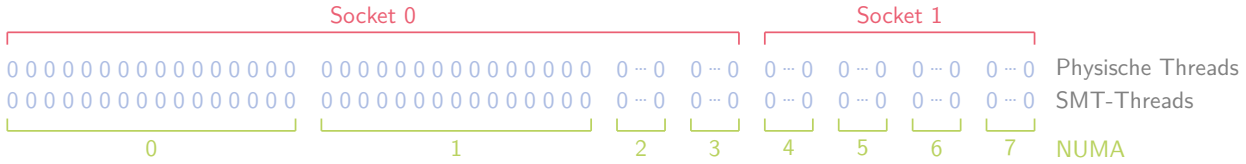

Abb. 2: Aufbau eines Knotens auf dem HPC-System JUSUF

Neben dem Betrieb mehrerer HPC-Systeme stellt das JSC auch Werkzeuge, Ressourcen und Wissen im HPC-Bereich für die Forscher von über 200 deutschen und europäischen Projekten bereit. [6].

1.3. Der Workload-Manager Slurm

Der Workload-Manager Slurm ist ein fehlertolerantes und hoch skalierbares Cluster-Management- und Job-Scheduling-System. Es wird weltweit auf Linux-Clustern unterschiedlicher Größe eingesetzt, darunter auch auf den Höchstleistungsrechnern des JSC. Es erfüllt auf diesen Systemen mehrere zentrale Funktionen: Es verwaltet Warteschlangen für anstehende Jobs, um den Wettbewerb um die begrenzten Ressourcen eines HPC-Systems zu organisieren, weist den Jobs für einen bestimmten Zeitraum spezifische Ressourcen zu und überwacht und steuert die Ausführung von Jobs auf diesen Ressourcen. [7, 8]

Eine Slurm-Installation umfasst mehrere Komponenten, darunter zwei zentrale Daemons[2] und mehrere Benutzerprogramme: Der `slurmctld`-Daemon dient als zentraler Manager von Slurm. Er überwacht verfügbare Ressourcen und plant die Abarbeitung von Jobs. Auf jedem Knoten eines HPC-Systems läuft zusätzlich der `slurmd`-Daemon, der die Prozesse der Nutzer verwaltet. Auf den Clustern des JSC wird dieser Daemon durch das Plug-in `psslurm` ersetzt, welches Slurm um den `psid`-Daemon, den Management-Daemon der Parastation Cluster Suite, erweitert. Über verschiedenste Benutzerprogramme, wie `srun` oder `sbatch`, kann ein Benutzer mit Slurm interagieren. Dazu zählt beispielsweise die Anforderung von Ressourcen für Programme oder das Ausführen dieser. Besonders wichtig ist das Benutzerprogramm `srun`, mit dem Anwendungen ausgeführt werden können. Damit bei der Ausführung von Anwendungen genügend Ressourcen zur Verfügung stehen, bietet `srun` mit Hilfe von verschiedenen Optionen den Nutzern die Möglichkeit, die Anzahl der benötigten Ressourcen zu spezifizieren. Dazu zählen beispielsweise die Optionen `--nodes`, `--ntasks` und `--cpus-per-task`, mit denen die Anzahl der benötigten Knoten, Prozesse und Threads pro Prozess festgelegt werden kann. Das folgende Code-Beispiel zeigt, wie der Befehl `srun` verwendet wird, um eine Anwendung auf einem Knoten auszuführen, welche insgesamt zehn Prozesse mit je drei Threads erzeugt. [8, 9]

[2] Hintergrundprozess, der ohne Interaktion mit Benutzern Aufgaben erfüllt

```
srun --nodes=1 --ntasks=10 --cpus-per-task=3 ./executable
```

Code-Beispiel 1: Beispiel für die Verwendung des Slurm-Befehls `srun`

1.3.1. Pinning

Pinning, auch als Prozess-Affinität bezeichnet, beschreibt die Bindung eines Prozesses oder Threads an bestimmte Kerne. Ein solcher gepinnter Prozess — ob als einzelner Prozess oder als Prozess mit mehreren Threads — kann ausschließlich auf den Kernen ausgeführt werden, an die er gebunden wurde. Dies verhindert unerwünschte Migration von Prozessen zwischen den Kernen eines Knotens und reduziert damit unter anderem aufwendige Fernspeicherzugriffe, wie sie beispielsweise in NUMA-Speicherarchitekturen auftreten können. Durch gezieltes Pinning können also effizientere Speicherzugriffe ermöglicht werden, was die Laufzeit von Anwendungen deutlich verkürzt und ihre Leistung enorm verbessert. Zusätzlich kann auch die Kommunikation zwischen Prozessen oder zwischen einem Prozess und einer GPU durch gezieltes Pinning optimiert werden, da Prozesse zum einen schneller mit GPUs, welche am selben Socket angeschlossen sind, und zum anderen schneller mit Prozessen im selben Socket kommunizieren können. [10, 11]

Slurm bietet verschiedene Möglichkeiten an, um das Pinning zu beeinflussen, wie zum Beispiel den `sbatch`-Befehl oder verschiedene Umgebungsvariablen. In dieser Arbeit konzentrieren wir uns aber auf den Befehl `srun`. Dieser Befehl stellt verschiedene Optionen zur Konfiguration des Pinnings zur Verfügung, welche in allen Slurm-Installationen standardmäßig verfügbar sind. Die konkrete Implementierung der Prozess-Affinität kann jedoch durch Plug-ins angepasst werden. Auf den HPC-Systemen des JSC wird dazu das Plug-in `psslurm` verwendet. Zu den verfügbaren Optionen des Befehls `srun` zählen unter anderem `--cpu-bind`, `--distribution` und `--threads-per-core`. Die Option `--cpu-bind` ermöglicht die Auswahl verschiedener Bindungstypen für das Pinning und die Option `--threads-per-core` gibt an, wie viele Hardware-Treads eines physischen Kerns maximal genutzt werden sollen. Mit Hilfe der Option `--distribution` kann die Verteilung der Prozesse und Threads auf mehreren Ebenen beeinflusst werden. Dazu zählt die Verteilung der Prozesse auf die Knoten, die Verteilung der Prozesse und Threads auf die Sockets innerhalb eines Knotens und die Verteilung der Prozesse und Threads auf die Kerne innerhalb eines Sockets. Das nachfolgende Beispiel zeigt, wie diese Option allgemein verwendet wird.

```
--distribution=<knoten-ebene>:<socket-ebene>:<kern-ebene>
```

Code-Beispiel 2: Allgemeine Verwendung der Option `--distribution` des Slurm Befehls `srun`

Die Verteilung der Prozesse auf die Knoten kann über den Wert `block` oder den Wert `cyclic` angepasst werden. Bei der Angabe des Wertes `block` wird jedem Knoten ein Block von aufeinanderfolgenden Prozesses zugewiesen, während die Prozesse bei der Angabe von `cyclic` zyklisch auf die Knoten verteilt werden. Auf den beiden anderen Ebenen gibt es ebenfalls die Werte `block` und `cyclic`. Auch auf der Socket- und der Kern-Ebene beeinflussen diese, ob Blöcke von aufeinanderfolgenden Prozessen auf die Sockets beziehungsweise Kerne verteilt werden oder ob diese Verteilung zyklisch abläuft. Zusätzlich steht der Wert `fcyclic` zur Verfügung, mit dem nicht nur die Prozesse, sondern auch die Threads innerhalb der Prozesse, zyklisch auf die Sockets oder Kerne verteilt werden. Insgesamt ergibt sich durch die Kombination dieser Optionen also ein sehr großer Parameterraum für das Pinning und Nutzer der HPC-Systeme können daher aus unzähligen verschiedenen Pinning-Konfigurationen wählen. [9, 10]

1.4. Pinning-Webtool

Das Pinning-Webtool ist eine interaktive, webbasierte Oberfläche, welche das Pinning auf verschiedenen Höchstleistungsrechnern des JSC simuliert. Sein Hauptziel ist es, den komplexen Vorgang des Pinnings auf möglichst verständliche und übersichtliche Weise darzustellen, damit Nutzer der Höchstleistungsrechner geeignete Pinning-Konfigurationen für ihre Anwendung finden oder die Auswirkungen einer bereits verwendeten Konfiguration besser nachvollziehen können.

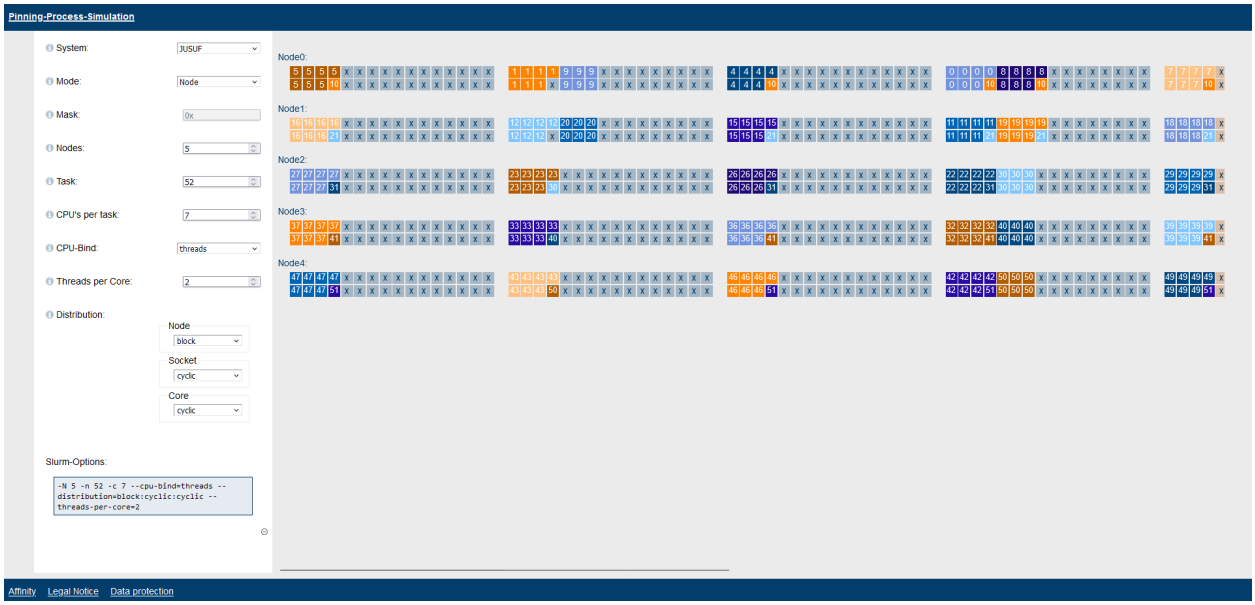

Abb. 3: Layout des Pinning-Webtools

Wie in Abbildung 3 zu sehen ist, gliedert sich das Layout des Webtools in zwei Bereiche, welche durch die Verwendung von HTML und CSS realisiert worden sind. Der linke Bereich ist für die Eingabe von Informationen durch den Benutzer vorgesehen. Er enthält ein Formular mit verschiedenen Eingabefeldern, über die ein Benutzer der Webseite alle benötigten Informationen, wie die benötigten

Ressourcen, das zu verwendende HPC-System und die zu nutzenden Pinning-Optionen, eingeben kann. In der Abbildung wurde beispielsweise das Pinning für eine Anwendung auf dem HPC-System JUSUF simuliert, die auf fünf Knoten ausgeführt wird und insgesamt 52 Prozesse mit je sieben Threads erzeugt. Für das Pinning wurden die Optionen `--threads-per-core=2`, `--cpu-bind=threads` und `--distribution=block:cyclic:cyclic` gewählt. Außerdem kann ausgewählt werden, wie das Pinning dargestellt werden soll. In der Abbildung wurde beispielsweise der Node-Modus gewählt. Auf der rechten Seite der Webseite ist das simulierte Pinning zu den gewählten Optionen zu sehen. Da in der Abbildung der Modus „Node“ gewählt wurde, werden auf der rechten Seite die fünf Knoten dargestellt, auf denen die Anwendung ausgeführt wird. Die verschiedenen kleinen Kästchen spiegeln dabei die logischen Kerne in den Knoten wieder. Anhand der Zahlen in den Kästchen, die je einen Prozess repräsentieren, ist zu erkennen, an welche logischen Kerne ein Prozess beim Pinning mit den ausgewählten Optionen gebunden wurde. Alle anderen Kästchen, die ein „x“ enthalten, wurden nicht verwendet.

Die Interaktivität dieser Webseite basiert auf einem Event, das ausgelöst wird, sobald ein Wert in einem der Eingabefelder geändert wird. Dieses Event wird von einem im Hintergrund laufenden JavaScript-Programm verarbeitet, welches die Darstellung des Pinnings aktualisiert. Sobald neue Parameter von einem Benutzer eingeben werden, liest das JavaScript-Programm im Hintergrund die aktuell eingegebenen Daten ein und berechnet im Anschluss das Pinning zu diesen Daten. Danach wird mit Hilfe von JavaScript eine SVG-Grafik, welche die Darstellung des Pinnings beinhaltet, erzeugt und in den rechten Bereich der bestehenden HTML-Seite eingebettet. [2, 12]

2. Entwicklungstechniken

Nachdem nun bekannt ist, was High-Performance Computing und vor allem Pinning ist, werden im Folgenden einige Grundlagen erläutert, welche für die zu Beginn der Arbeit beschriebene Automatisierung der Pinning-Vergleiche mittels CI/CD benötigt werden. Dazu zählen zum einen Grundlagen zur Automatisierung mit CI/CD in GitLab und zum anderen einige Aspekte der Webentwicklung.

2.1. CI/CD in GitLab

Unter Continuous Integration (CI) und Continuous Delivery/Deployment (CD) versteht man eine Methode der Softwareentwicklung, bei der der gesamte Entwicklungsprozess einer Software von der Erstellung bis hin zur Veröffentlichung optimiert wird, indem Änderungen kontinuierlich integriert, getestet und veröffentlicht werden. Dies führt dazu, dass Fehler in der Software frühzeitig festgestellt und behoben werden können. [13]

CI/CD soll im Anschluss zur Automatisierung der Pinning-Vergleiche verwendet und in GitLab konfiguriert werden. Die Grundlage für die Konfiguration von CI/CD in GitLab stellt eine YAML-Datei mit dem Namen `.gitlab-ci.yml` dar. Innerhalb dieser Datei werden die benötigten CI/CD-Pipelines definiert. Eine solche Pipeline besteht aus einer Reihe von Stages und Jobs, welche automatisch ausgeführt werden sollen. Eine Stage ist im Grunde ein Arbeitsschritt, in dem mehrere verschiedene Operationen ausgeführt werden sollen. Sie beinhaltet die Jobs, welche die eigentlichen Operationen enthalten. Die Operationen innerhalb eines Jobs werden dabei sequentiell ausgeführt, während die verschiedenen Jobs innerhalb einer Stage parallel ausgeführt werden. Die verschiedenen Stages werden mit dem Schlüsselwort `stages` definiert. Die Reihenfolge der Definitionen innerhalb des Schlüsselworts `stages` gibt die Ausführungsreihenfolge der definierten Stages an.

```yaml
stages:
    - build

build-job:
    stage: build
    script:
        - echo "This job runs in the build stage."
```

Code-Beispiel 3: Definition einer simplen CI/CD-Pipeline in GitLab

Zur Definition eines Jobs werden, wie in Code-Beispiel 3, unter anderem die Schlüsselworte `stage` und `script` verwendet, mit denen man festlegen kann, in welcher Stage ein Job laufen und welche Operationen ausgeführt werden sollen. Außerdem erhält jeder Job einen Namen. Im vorherigen Beispiel heißt der definierte Job beispielsweise `build-job`. Kommt es in einem Job zu Fehlern, wird die Pipeline abgebrochen und Jobs aus späteren Stages werden nicht mehr gestartet. Außerdem werden die Entwickler automatisch per Email über den Abbruch der Pipeline informiert. [14]

2.1.1. Gitlab Runner

GitLab Runner ist eine Anwendung, die benötigt wird, um Jobs innerhalb einer Pipeline auszuführen. Sie wird auf einem physischen Rechner oder einer virtuellen Instanz installiert und in ihr können mehrere Runner, also Prozesse, welche die Jobs letztendlich ausführen, registriert werden. Bei dieser Registrierung wird die Kommunikation zwischen GitLab und der Maschine, auf der GitLab Runner installiert ist, eingerichtet. Außerdem wird die Ausführungsumgebung der Jobs, wie beispielsweise ein Docker-Container oder eine Shell-Umgebung, festgelegt und es werden sogenannte Tags definiert, durch die ein spezieller Runner identifiziert werden kann. Während der Definition eines Jobs in der `gitlab-ci.yaml`-Datei werden genau diese Tags benötigt, um einen Runner auszuwählen, wodurch eine Pipeline sehr flexibel gestaltet und verschiedenste Arten von Ausführungsschritten automatisiert durchgeführt werden können. Zur Auswahl eines Runners müssen über das `tags`-Schlüsselwort Tags angegeben werden, über die ein Runner eindeutig identifiziert werden kann. Im GitLab des JSC ist beispielsweise ein Docker-Runner registriert, welcher über das Tag `public-docker`, wie in Code-Beispiel 4 zu sehen ist, identifiziert werden kann. [15, 16]

```
job:
    tags:
        - public-docker
    script:
        - echo "This job is run by the docker-runner."
```

Code-Beispiel 4: Verwendung von Tags zur Identifizierung eines Runners in GitLab

2.1.2. Artefakte

In GitLab können Jobs ein Archiv mit Dateien und Verzeichnissen erzeugen, welche als Job-Artefakte bezeichnet werden. Diese Artefakte werden verwendet, um Daten an spätere Jobs weiterzugeben, da jeder Job standardmäßig alle Artefakte von vorherigen Stages lädt und somit auf diese zugreifen kann. Artefakte werden, wie in Code-Beispiel 5 zu sehen ist, mit dem Schlüsselwort `artifacts` definiert.

```
job:
    script:
        - echo "Job with artifacts"
    artifacts:
        paths:
            - file.pdf
            - dir_name/
        expire_in: 1 week
```

Code-Beispiel 5: Verwendung von Job-Artefakten in GitLab

Unter dem Schlüsselwort `paths` können alle Dateien und Verzeichnisse angebenen werden, die als Artefakt gespeichert werden sollen. Demnach werden in Code-Beispiel 5 zum einen die Datei `file.pdf` und zum anderen der Ordner `dir_name` als Artefakt gespeichert. Die Pfade zu diesen Dateien und Verzeichnissen müssen relativ zum Hauptverzeichnis des GitLab-Repositories angegeben werden. Zusätzlich kann mit dem Schlüsselwort `expire_in` angegeben werden, wie lange erzeugte Artefakte gespeichert werden sollen. Dadurch können Artefakte automatisch gelöscht werden, was vor allem bei sehr großen Datenmengen relevant ist. Die aktuellsten Artefakte, sowie die Artefakte der letzten erfolgreichen Pipeline werden jedoch unabhängig von dieser Angabe nicht gelöscht. [17]

2.1.3. Gitlab Pages

Mit GitLab Pages können statische Webseiten, also Webseiten, die nicht auf serverseitige Prozesse angewiesen sind, direkt aus einem GitLab-Repository heraus veröffentlicht werden, indem GitLab CI/CD verwendet wird. Hierfür wird in der `.gitlab-ci.yml`-Datei ein Job mit dem Namen `pages` definiert, der GitLab signalisiert, dass eine Webseite generiert werden soll. Ein Beispiel für die Definition dieses Jobs ist in Code-Beispiel 6 zu sehen. Alternativ kann GitLab auch über die Angabe `pages: true` innerhalb eines Jobs mitgeteilt werden, dass eine Webseite generiert werden soll. Da GitLab die Webseite mit den Dateien aus einem spezifischen Ordner namens `public` erstellt, wird dieser Job standardmäßig dazu verwendet alle benötigten Dateien in diesen Ordner zu verschieben. [18]

```
pages:
    script:
        - echo "Preparing the directory public"
    artifacts:
        paths:
            - public/
```

Code-Beispiel 6: Verwendung des `pages`-Jobs zur Veröffentlichung einer Webseite mit GitLab Pages

2.2. Jacamar CI

In GitLab ist es nicht ohne weiteres möglich, innerhalb einer CI/CD-Pipeline Jobs auf Höchstleistungsrechnern auszuführen. Dies ist jedoch während der Automatisierung der Pinning-Vergleiche notwendig, um das Pinning auf den HPC-Systemen auszulesen. Damit dennoch Jobs auf den Höchstleistungsrechnern ausgeführt werden können, wird das Open-Source-Projekt Jacamar CI verwendet, welches eine Schnittstelle zwischen GitLab CI/CD-Pipelines und HPC-Ressourcen bietet. Um sicherzustellen, dass nur autorisierte Nutzer Zugriff auf HPC-Ressourcen erhalten, verwendet Jacamar CI einen Authentifizierungsmechanismus, der auf einem Abgleich der Benutzernamen zwischen GitLab und den HPC-Systemen basiert.

Innerhalb des GitLabs des JSC sind gemeinsam genutzte Jacamar-CI-Runner eingerichtet, um über GitLab Zugang zu den HPC-Systemen des JSC zu erhalten. Auf jedem Höchstleistungsrechner sind zwei verschiedene Runner verfügbar. Dazu zählt ein Slurm-Runner, welcher mit Hilfe von Slurm Jobs startet, und ein Shell-Runner, der auf den Login-Knoten der HPC-Systeme arbeitet und daher beispielsweise für die Vorbereitung von ausführbaren Programmen ideal ist. Jeder dieser Runner wird über eindeutige Tags identifiziert, damit er einfach in Pipelines verwendet werden kann. [19]

2.3. HTML

Die Auszeichnungssprache HTML, welche auf SGML[3] basiert, wurde im Jahr 1990 von Tim Berners-Lee entwickelt. Die Abkürzung HTML steht für HyperText Markup Language. Mit einer solchen Auszeichnungssprache können die Bestandteile eines textorientierten Dokuments, wie beispielsweise Überschriften, Listen oder Tabellen, beschrieben werden, wobei HTML standardmäßig dazu verwendet wird die Struktur von Webseiten zu beschreiben. Sie beschreibt also die inhaltliche Gliederung einer Webseite und nicht das Aussehen dieser, da das Aussehen einer Webseite nicht mit HTML, sondern mit CSS, kurz für Cascading Style Sheets, definiert wird. Später HTML wird dazu verwendet, die Struktur der Webseite zu beschreiben, welche die Ergebnisse der Pinning-Vergleiche darstellen soll.

Die Definition der Webseiten-Struktur wird über verschiedene HTML-Auszeichnungselemente realisiert, die einem Internet-Browser mitteilen, wie der Inhalt angeordnet werden soll. Ein solches HTML-Element wird, wie in Code-Beispiel 7 zu sehen ist, über drei Teile definiert: ein Start-Tag, einen Inhalt und ein End-Tag.

```
<tagname> Inhalt... </tagname>
```

Code-Beispiel 7: Aufbau eines HTML-Elements

[3] SGML ist eine Metasprache, also eine Werkzeug für die Definition anderer Sprachen, die 1986 standardisiert wurde [20]

Der Grundaufbau eines HTML-Dokuments wird anhand des folgenden Beispiels erläutert:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <h1>My First Heading</h1>
        <p>My first paragraph.</p>
    </body>
</html>
```

Code-Beispiel 8: Aufbau eines HTML-Dokuments

Jedes HTML-Dokument sollte mit der <!DOCTYPE html>-Deklaration beginnen, da diese das Dokument als ein HTML-Dokument kennzeichnet. Das folgende <html>-Element ist das Grundelement eines HTML-Dokumentes. Es umschließt alle weiteren verwendeten HTML-Elemente. Der Bereich im <html>-Element unterteilt sich in die zwei Teilbereiche <head> und <body>. Das <head>-Element beinhaltet alle Meta-Daten der Webseite, wie beispielsweise den Titel, der in der Registerkarte der Webseite angezeigt wird, während das <body>-Element alle sichtbaren Inhalte der Webseite, wie Bilder, Überschriften oder Absätze, enthält. [21–23]

2.4. Javascript

JavaScript ist eine Programmiersprache, welche im Jahr 1995 von der Firma Netscape entwickelt und lizenziert wurde. Sie wird als Zusatztechnik in Webseiten eingebunden, um für eine Webseite neben der Struktur und dem Aussehen, was mit Hilfe von HTML und CSS implementiert wird, auch eine Verhaltenssschicht einzuführen. Dadurch können Webseiten dynamisch auf Benutzerinteraktionen reagieren und das HTML-Dokument verändern, was auch für die Webseite relevant ist, die später die Ergebnisse der Pinning-Vergleiche darstellen soll.

Ursprünglich war JavaScript eine reine Interpretersprache, welche direkt im Browser des Benutzers ausgeführt werden konnte. Genau dazu wird JavaScript beispielsweise auch im zuvor vorgestellten Pinning-Webtool verwendet. Durch Entwicklungen in den letzten Jahren können heute jedoch auch serverseitige Anwendungen mit JavaScript geschrieben werden. Damit hat sich JavaScript von einer rein browserbasierten Technik zu einer vielfältig einsetzbaren Sprache entwickelt, welche im Anschluss unter anderem dazu verwendet wird, die Pinning-Vergleiche durchzuführen. [23]

2.4.1. Document Object Model

Das Document Object Model, kurz DOM, stellt eine Schnittstelle zwischen JavaScript und HTML-Dokumenten dar. Beim Parsen eines HTML-Dokuments übersetzt der Internet-Browser die textbasierte Struktur des Dokuments in eine Baumstruktur, den sogenannten DOM-Baum, in dem jedes HTML-Element als ein Baum-Knoten dargestellt wird. Beispielsweise sieht ein DOM-Baum für das HTML-Dokument aus Code-Beispiel 8 wie folgt aus:

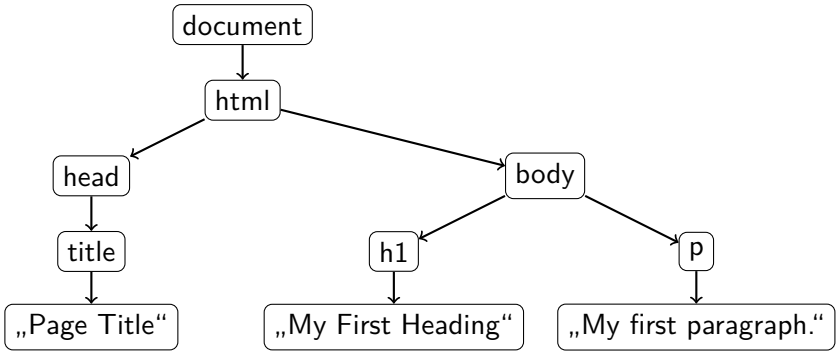

Abb. 4: DOM-Baumstruktur des HTML-Dokuments aus Code-Beispiel 8

In JavaScript kann über ein globales `document`-Objekt und mit Hilfe von Methoden wie `getElementById` oder `querySelector`, die von der DOM-API zur Verfügung gestellt werden, auf Elemente des DOM-Baums zugegriffen werden. Dadurch können HTML-Dokumente bzw. DOM-Bäume mit Hilfe von JavaScript und dem Document Object Model verändert oder sogar neu erstellt werden, was in der modernen Webentwicklung unverzichtbar geworden ist. Ein Beispiel hierfür ist auch wieder das Pinning-Webtool, in dem mit Hilfe der DOM-API auf die eingegebenen Daten der Nutzer zugegriffen und die erstellte SVG-Datei in das HTML-Dokument eingebettet wird. [24, 25]

3. Verwendete Software-Pakete

Nachdem im vorherigen Kapitel die Grundlagen zu CI/CD und Webentwicklung erläutert wurden, werden im Folgenden die Software-Pakete Node.js und JUBE sowie ein kleines C++-Programm vorgestellt, die während der Pipeline zur Automatisierung der Pinning-Vergleiche verwendet werden.

3.1. Node.js

Node.js ist eine Open-Source-JavaScript-Laufzeitumgebung, welche plattformübergreifend verwendet werden kann, um insbesondere hochperformante und skalierbare Netzwerkanwendungen zu entwickeln. Eine Anwendung, die mit Node.js ausgeführt wird, benötigt nur einen einzigen Prozess und kommt ohne zusätzliche Threads aus. Dies wird durch die Bereitstellung von effizienten und asynchronen Funktionen in der Standardbibliothek und in externen Bibliotheken ermöglicht. Wird mit diesen Funktionen eine I/O-Operation, wie der Zugriff auf das Dateisystem oder das Lesen aus dem Netzwerk, durchgeführt, so wird der Thread nicht blockiert, was bedeutet, dass der Prozess auch bei aufwendigen Operationen keine Wartezeiten verursacht. Stattdessen kann Node.js parallel weitere Operationen bearbeiten, wodurch mehrere gleichzeitige Anfragen mit minimalem Aufwand verarbeitet werden können. Sobald von einer gestarteten asynchronen Operation eine Antwort zurückkommt, wird diese Operation wieder aufgenommen und fortgesetzt.

Die Ausführung von Anwendungen erfolgt in der Konsole über den `node`-Befehl. Dieser erwartet die auszuführende Datei als Parameter und kann jede JavaScript-Datei starten, unabhängig davon, ob sie für Netzwerkanwendungen vorgesehen ist. Dadurch ist Node.js nicht nur für die Entwicklung von Netzwerkanwendungen geeignet, sondern auch für andere Anwendungsbereiche. Im Anschluss wird Node.js dazu verwendet, ein JavaScript-Programm innerhalb einer CI/CD-Pipeline auszuführen. [26, 27]

3.2. JUBE

JUBE ist eine Benchmarking- und Workflow-Umgebung, welche aktiv vom JSC entwickelt wird und nachfolgend verwendet wird, um das Pinning auf den HPC-Systemen für zahlreiche Pinning-Konfigurationen auszulesen. Der Schwerpunkt der Software liegt dabei auf der Verwaltung, Durchführung und Reproduzierbarkeit von komplexen Benchmarks beziehungsweise Workflows. Sie bietet in einem skriptbasierten Rahmen zahlreiche Möglichkeiten zur einfachen Erstellung von Workflow- und Benchmark-Sets, zur

Ausführung dieser Sets auf verschiedenen Computersystemen und zur Auswertung von Ergebnissen. Obwohl JUBE in erster Linie für den Einsatz auf Höchstleistungsrechnern konzipiert ist, funktioniert es auch auf herkömmlichen Rechnern mit Linux- oder MacOS-Betriebssystemen und ist dadurch vielseitig einsetzbar. [28, 29]

JUBE unterstützt sowohl XML-basierte als auch YAML-basierte Dateien als Eingabe-Skripte, in denen die Benchmarks oder Workflows konfiguriert werden. Beide Formate unterstützen den gleichen Umfang an Funktionalität. Intern werden YAML-basierte Eingabedateien jedoch aus historischen Gründen in XML konvertiert, bevor sie von JUBE weiterverarbeitet werden. Aus diesem Grund wird die Verwendung von JUBE im Folgenden ausschließlich anhand von XML-Dateien gezeigt.

Das nachfolgende Beispiel zeigt den grundsätzlichen Aufbau eines JUBE-Skripts:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
    <benchmark name="hello_world" outpath="bench_run">

        <!-- Configuration -->
        <parameterset name="hello_parameter">
            <parameter name="hello_str" tag="de">Hallo Welt</parameter>
            <parameter name="hello_str" tag="en">Hello World</parameter>
        </parameterset>

        <!-- Operation -->
        <step name="say_hello">
            <use>hello_parameter</use> <!-- use existing parameterset -->
            <do>echo $hello_str</do> <!-- shell command -->
        </step>
    </benchmark>
</jube>
```

Code-Beispiel 9: Aufbau einer einfachen XML-basierten Eingabedatei von JUBE

Jedes XML-basierte JUBE-Skript beginnt mit dem Tag `<jube>`. Innerhalb dieses Tags werden die Benchmarks konfiguriert. Das nächste Tag lautet `<benchmark>`. In diesem werden alle relevanten Konfigurationen für einen Benchmark definiert. Dazu gehören beispielsweise Definitionen von Parametern oder den auszuführenden Operationen, den sogenannten Steps. Ein Parameter kann als Variable interpretiert werden. Er speichert Daten, welche innerhalb des Benchmarks benötigt werden. Dabei kann ein Parameter nicht nur einen Wert enthalten, sondern es können mehrere Werte angegeben werden. Durch die Angabe mehrerer Parameter entstehen eine Vielzahl von Parameterkombinationen, welche den

sogenannten Parameterraum bilden. Ein Step ist vereinfacht gesagt eine Liste von Shell-Operationen, die für jede mögliche Kombination der angegebenen Parameter einmal ausgeführt wird. Über Tagging, also das Angeben eines tag-Attributs in verschiedenen JUBE-Tags, können spezifische Definitionen bei der Ausführung von Benchmarks inkludiert oder exkludiert werden. [30, 31]

Ausgeführt wird ein JUBE-Skript beispielsweise über den Befehl `jube run`. Dieser gilt als erfolgreich beendet, sobald alle Shell-Befehle der Steps ausgeführt wurden. Auf den Höchstleistungsrechnern bedeutet dies, dass alle Jobs in die Warteschlange von Slurm eingestellt, jedoch nicht zwingend beendet wurden. Mit der Option `--tag` können Tags angegeben werden, welche beim Tagging verwendet wurden. Führt man das JUBE-Skript aus Code-Beispiel 9 mit folgendem Befehl aus und gibt das Tag `de` an, so erhält der Parameter `hello_str` den Wert `Hallo Welt` und es wird „Hallo Welt“ ausgegeben und nicht „Hello World“.

```
jube run hello_world.xml --tag de
```

Code-Beispiel 10: Ausführung des JUBE-Skripts aus Code-Beispiel 9 mit dem `jube run` Befehl

Wenn die Ausführung eines Benchmarks oder Workflows mit JUBE vollständig automatisiert werden soll, sollte jedoch der Befehl `jube-autorun` verwendet werden, da dieser Befehl erst erfolgreich beendet wird, sobald alle gestarteten Jobs beendet wurden. Er ruft intern den `jube run` Befehl auf und wartet anschließend in einer Schleife darauf, dass alle gestarteten Jobs beendet werden. Dem `jube-autorun`-Befehl können über die Option `-r` die verschiedenen Argumente des `jube run`-Befehls mitgegeben werden. Das folgende Beispiel zeigt, wie `jube-autorun` verwendet wird, um ebenfalls das JUBE-Skript aus Code-Beispiel 9 auszuführen. [31, 32]

```
jube-autorun -r "--tag de" hello_world.xml
```

Code-Beispiel 11: Ausführung des JUBE-Skripts aus Code-Beispiel 9 mit dem `jube-autorun` Befehl

3.3. C++-Programm zum Auslesen des Pinnings

Während der später implementierten CI/CD-Pipeline wird auf ein bereits implementiertes C++-Programm zurückgegriffen, das im Anhang in Code-Beispiel 19 zu finden ist. Dieses Programm, welches ursprünglich ein mit MPI und OpenMP parallelisiertes Hello-World-Programm war, wurde so erweitert, dass das Pinning für alle Prozesse und Threads ausgelesen wird. Es liest für jeden Thread aus auf welchem Kern und welchem Knoten dieser ausgeführt wird und schreibt die ermittelten Daten im Anschluss auf die Standardausgabe.

```
0     0    48 jrtc0009 ./compile/hello_world.exe
0     1   176 jrtc0009 ./compile/hello_world.exe
0     2    49 jrtc0009 ./compile/hello_world.exe
0     3   177 jrtc0009 ./compile/hello_world.exe
1     0    16 jrtc0009 ./compile/hello_world.exe
1     1   144 jrtc0009 ./compile/hello_world.exe
1     2    17 jrtc0009 ./compile/hello_world.exe
1     3   145 jrtc0009 ./compile/hello_world.exe
```

Abb. 5: Ausgabe des C++-Programms, welches das Pinning ausliest

Wie in Abbildung 5 zu sehen ist, enthält jede ausgegebene Zeile fünf verschiedene Werte. Die ersten beiden Werte geben die Prozess- und Thread-ID an und anhand der folgenden Werte kann abgelesen werden, an welchen Kern der entsprechende Thread gebunden worden ist. Dabei gibt der dritte Wert den für diesen Thread verwendeten Kern und der vierte Wert den verwendeten Knoten an. Der letzte Wert in jeder Zeile enthält den relativen Pfad zu der ausführbaren Datei des Programms und ist daher für das Pinning irrelevant.

4. Entwicklungsschritte

Nachdem nun alle benötigten Grundlagen und Software-Pakete vorgestellt worden sind, soll nun erläutert werden, wie eine CI/CD-Pipeline erstellt werden kann, welche die Pinning-Vergleiche bzw. die Schritte automatisiert, die zu Beginn dieser Arbeit beschrieben wurden. Abbildung 6 verdeutlicht, wann die zuvor beschriebenen Techniken und vorgestellten Software-Pakete in der Pipeline verwendet werden.

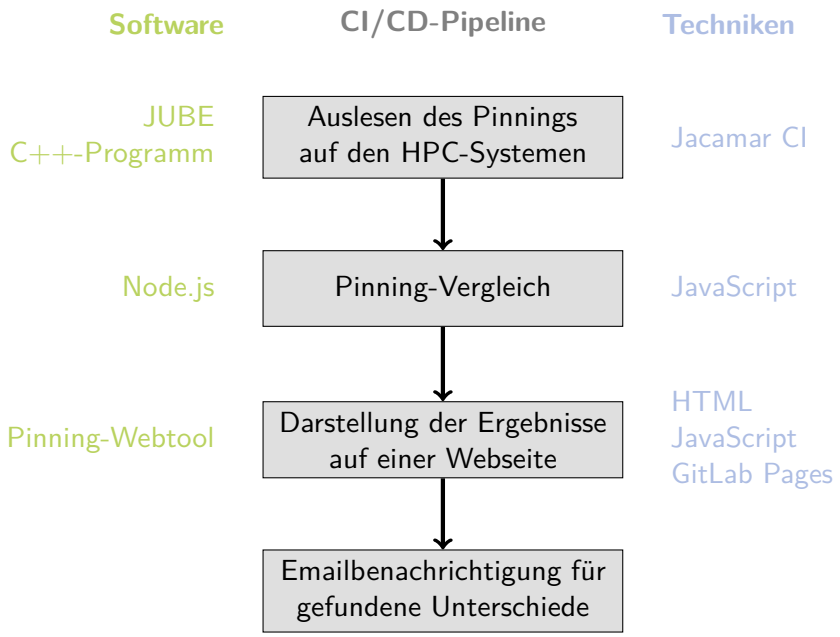

Abb. 6: Verwendung der vorgestellten Techniken und Software in der zu implementierenden CI/CD-Pipeline

4.1. Manuelle Ausführung

Die Grundlage für die spätere Automatisierung des Vergleichs in einer CI/CD-Pipeline ist die manuelle Durchführung aller erforderlichen Schritte, um die Funktionalität jedes Schrittes zu validieren und notwendige Anpassungen für die Integration in die Pipeline vorzunehmen.

4.1.1. Auslesen des Pinnings auf den HPC-Systemen

Um das Pinning auf den HPC-Systemen auslesen zu können, wurde das zuvor vorgestellte C++-Programm verwendet, welches das Pinning ausliest und dieses strukturiert formatiert auf die Standardausgabe schreibt. Mit Hilfe eines bereits vorhandenen JUBE-Skriptes wird dieses Programm wiederholt für verschiedene Pinning-Konfigurationen aufgerufen. Genauer gesagt startet das JUBE-Skript das C++-Programm für

jede mögliche Parameter-Kombination eines Parameterraums, welcher über die Angabe verschiedener Tags beim Starten des JUBE-Skriptes ausgewählt worden ist. Die Ausgabe des C++-Programms wird jeweils in einer Datei gespeichert, deren Name die jeweiligen genutzten Pinning-Optionen beinhaltet, und am Ende werden alle erzeugten Dateien in einem Ordner namens `pin_logs` gesammelt. Dieses JUBE-Skript wurde mit Hilfe des Befehls `jube-autorun` getestet, da dieser auf die Beendigung von Jobs wartet.

```
jube-autorun -r "-t js gp cb_all cd_all" all_hello_reduced.xml
```

Code-Beispiel 12: Verwendeter Befehl zum Auslesen des Pinnings für verschiedene Pinning-Konfigurationen auf dem HPC-System JUSUF

Um dabei einen möglichst großen Parameterraum abzudecken, wurden, wie in Code-Beispiel 12 zu sehen, die Tags `cb_all` und `cd_all` verwendet, da diese bereits die Werte `threads`, `rank`, `rank_ldom` und `cores` der Option `--cpu-bind` sowie die Werte `cyclic` und `fcyclic` der Distribution auf Kernebene beinhalten.

4.1.2. Durchführung des Vergleichs

Nachdem das Auslesen des Pinnings auf den HPC-Systemen fehlerfrei funktionierte, konnte dieses mit dem simulierten Pinning des Webtools verglichen werden. Hierzu wurde ein kurzes JavaScript-Programm implementiert, welches den Namen des Ordners, in dem sich die Dateien mit dem ausgelesenen Pinning befinden, als Kommandozeilen-Argument erhält. Dieses Programm iteriert über alle Dateien in diesem Ordner, liest die verwendeten Pinning-Optionen aus den Dateinamen sowie das Pinning selbst aus den Dateien aus und berechnet basierend auf den Pinning-Optionen das simulierte Pinning. Dazu wird auf bereits im Pinning-Webtool implementierte Methoden zurückgegriffen, die die Berechnung des Pinnings durchführen. Im Anschluss werden beide Pinning-Varianten verglichen, wobei es zu drei verschiedenen Ergebnissen kommen kann: Die beiden Pinning-Varianten können identisch sein, sie können Unterschiede aufweisen oder der Vergleich kann nicht durchgeführt werden, weil es keine Pinning-Simulation für die verwendeten Optionen gibt. Abhängig vom Ergebnis wird der Dateiname in eines von drei Arrays geschrieben, welche am Ende des Programms in einer JSON-Datei abgespeichert werden, damit die Ergebnisse auch nach dem Programmende noch verfügbar sind. Damit die Entwickler in der später definierten Pipeline automatisiert per Email über bestehende Unterschiede im Pinning informiert werden können, wird das Programm außerdem mit dem Exit-Code 1 beendet, falls Unterschiede auftreten. Dies führt nämlich dazu, dass die spätere Pipeline abgebrochen wird und GitLab die Entwickler automatisiert per Email über den Abbruch informiert, wodurch diese implizit auf die gefundenen Unterschiede hingewiesen werden. Die Ausführung des Programms wurde, wie in Code-Beispiel 13 zu sehen ist, mit Node.js getestet.

```
node comparison.js pin_logs/
```

Code-Beispiel 13: Verwendung von Node.js zur Ausführung des Pinning-Vergleichs

4.1.3. Erstellung einer Webseite für die Darstellung der Vergleichsergebnisse

Zum Schluss sollten die Ergebnisse des Vergleichs auf einer Webseite dargestellt werden, um die Entwickler des Pinning-Webtools bei der Analyse der Ergebnisse zu unterstützen, sodass die Ursachen für entstandene Unterschiede möglichst schnell gefunden werden können. Dazu wurde die bereits vorhandene Implementierung des Pinning-Webtools größtenteils übernommen, da diese als statische Webseite auch für die Verwendung mit GitLab Pages geeignet ist. Kleinere Anpassungen wurden nur in der Struktur der Webseite, die mit HTML beschrieben wurde, und dem Verhalten vorgenommen, das in JavaScript implementiert wurde. Im Aufbau der Webseite wurde auf der rechten Seite, die das Pinning darstellt, ein weiterer Bereich ergänzt, damit nun das tatsächliche Pinning auf dem HPC-System (REAL SITUATION) und das simulierte Pinning (RULE-BASED) gleichzeitig angezeigt werden können. Außerdem wurde das Formular auf der linken Seite der Webseite angepasst, sodass Nutzer nun neben dem Modus nur eine Kategorie von Ergebnissen sowie eine Datei, die angezeigt werden soll, auswählen können. Die neue Struktur der Webseite sieht folgendermaßen aus:

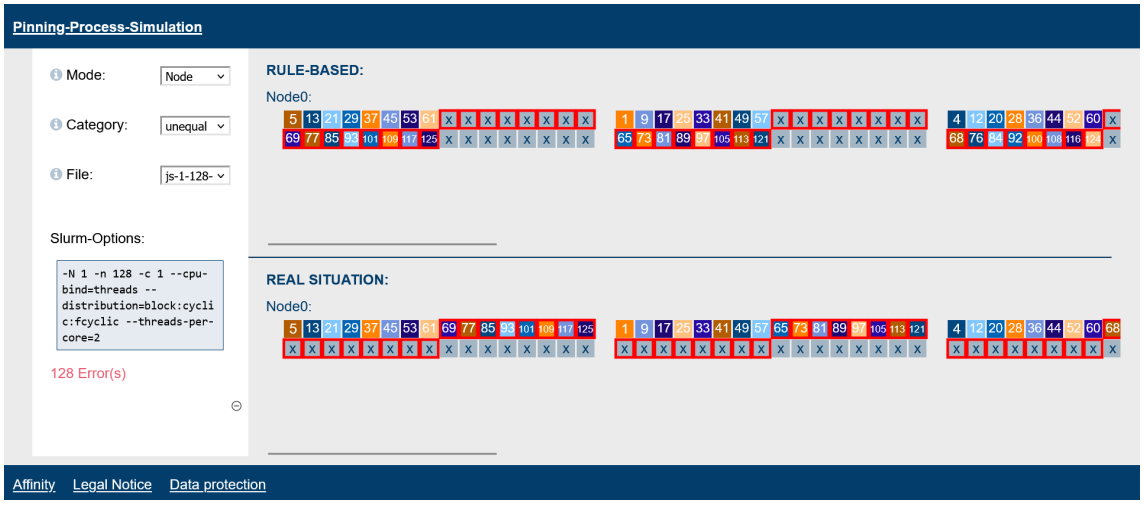

Abb. 7: Webseite, die die Ergebnisse des Pinning-Vergleichs für einen fiktiven Fall darstellt

Das im Hintergrund ablaufende JavaScript-Programm wurde ebenfalls angepasst, damit nun neben der simulierten Pinning-Darstellung auch die Darstellung für das Pinning auf den HPC-Systemen generiert werden kann. Dazu wurden neue EventListener ergänzt, die auf die Änderungen der verschiedenen Elemente im HTML-Formular reagieren, und die Methoden zur Darstellung des Pinnings wurden so angepasst, dass diese nun beide Darstellungen für das Pinning erzeugen und zusätzlich die bestehenden Unterschiede markieren, indem die betroffenen Stellen durch rote Umrandungen hervorgehoben werden.

4.2. Automatisierung mit GitLab CI/CD

Zur Automatisierung der zuvor manuell ausgeführten Schritte wurde in GitLab eine CI/CD-Pipeline implementiert. Dazu wurden zuerst die Stages definiert, die benötigt werden, um die einzelnen zuvor manuell ausgeführten Schritte in der Pipeline abzubilden. Daher wurde die Stage `generate_files` für die Erzeugung der Pinning-Dateien, die Stage `compare` für den Vergleich und die Stage `deploy` für die Veröffentlichung der Webseite festgelegt. Da ein Abbruch der Pipeline im Falle von gefundenen Unterschieden erforderlich ist, damit die Entwickler des Tools per Email informiert werden, die Webseite jedoch in jedem Fall generiert werden soll, wurde zusätzlich noch eine weitere Stage namens `result` ergänzt. Diese entscheidet nach der Veröffentlichung der Webseite, ob die Pipeline erfolgreich durchläuft oder abgebrochen wird. Insgesamt umfasst die Pipeline somit die vier in Code-Beispiel 14 definierten Stages.

```
stages:
    - generate_files
    - compare
    - deploy
    - result
```

Code-Beispiel 14: Definition der endgültigen Stages für die Pinning-Verifikation

Damit in den vier Stages Operationen ausgeführt werden können, musste im Anschluss für jede Stage mindestens ein Job definiert werden. In der Stage `generate_files` sollte dieser Job, die Generierung der Dateien, die das ausgelesene Pinning enthalten, auf den HPC-Systemen übernehmen, wobei zunächst nur Dateien auf dem Höchstleistungsrechner JUSUF erzeugt werden sollten. Da es in GitLab jedoch nicht ohne weiteres möglich ist, während einer CI/CD-Pipeline Jobs auf einem Höchstleistungsrechner auszuführen, musste hier Jacamar-CI verwendet werden. Genauer gesagt wurde der auf JUSUF verfügbare Shell-Runner von Jacamar CI verwendet, da für die Generierung der Dateien die Software JUBE benötigt wird und diese auf einem Login-Knoten in der Shell ausgeführt wird. Dieser Runner hat die Tags `jusuf`, `login`, `jacamar` und `shell`. Er kann jedoch über die Tags `jusuf` und `login` eindeutig identifiziert werden, weshalb in Code-Beispiel 15 nur diese beiden Tags angegeben wurden. Um sich mit GitLab bei Jacamar CI zu authentifizieren wird außerdem ein JSON Web Token benötigt, welches mit dem Schlüsselwort `id_tokens` angelegt wurde. Um die Dateien letztendlich zu erzeugen, wurde das JUBE-Skript, das die Pinning-Dateien erzeugt, wie zuvor bei der manuellen Ausführung gestartet, wozu jedoch mit dem Befehl `ml` zuerst JUBE als Modul geladen werden musste. Zum Schluss war es dann noch notwendig den Ordner `pin_logs`, indem die erzeugten Dateien gesammelt wurden, als Artefakt zu speichern, damit er in späteren Stages verfügbar ist.[14]

```yaml
generate_files_jusuf:
    id_tokens:
        SITE_ID_TOKEN:
            aud: "https://gitlab.jsc.fz-juelich.de"
    stage: generate_files
    tags:
        - jusuf
        - login
    script:
        - ml JUBE
        - jube-autorun -r "--tag js gp cb_all cd_all" all_hello_reduced.xml
    artifacts:
        paths:
            - pin_logs/
        expire_in: "1 hour"
```

Code-Beispiel 15: Erzeugung der Pinning-Dateien in der CI/CD-Pipeline

In der Stage compare wurde ebenfalls ein Job benötigt, um Operationen auszuführen. Dieser ist dafür verantwortlich den Vergleich zwischen dem auf dem HPC-System JUSUF ausgelesenen Pinning und dem simulierten Pinning im Pinning-Webtool durchzuführen. Da in diesem Job das zuvor für den Vergleich implementierte JavaScript-Programm, wie auch bei der manuellen Ausführung, mit Node.js ausgeführt werden soll, wurde ein Node.js-Docker-Image benötigt. Damit dieses Image mit dem Schlüsselwort image geladen werden konnte, wurde der vom JSC bereitgestellte Docker-Runner verwendet. Ein Problem stellte

```yaml
compare_pinning:
    stage: compare
    image: node:latest
    tags:
        - public-docker
    script:
        - node comparison.js pin_logs/ || echo $? > exit_code.txt
    artifacts:
        paths:
            - results.json
            - exit_code.txt
        expire_in: "1 hour"
```

Code-Beispiel 16: Vergleich Pinnings in der CI/CD-Pipeline

jedoch noch der Exit-Code des JavaScript-Programms dar. Falls beim Vergleich Unterschiede aufgefallen wären und das Programm mit dem Exit-Code 1 geendet hätte, wäre die Pipeline abgebrochen worden, bevor die Webseite mit den Ergebnissen hätte veröffentlicht werden können. Damit diese in jedem Fall veröffentlicht wird, war es notwendig den Error-Code des Programms in eine Datei umzulenken, sodass

dieser, falls Unterschiede gefunden wurden, nicht den Abbruch der CI/CD-Pipeline auslöst. Die Umleitung des Error-Code ist unter anderem in Code-Beispiel 16 zu sehen. Da sowohl die Datei mit dem Error-Code als auch die Datei mit den Vergleichsergebnissen in späteren Stages gebraucht werden, mussten diese, wie zuvor der Ordner `pin_logs`, als Artefakt gespeichert werden.

Der Job in der dritten Stage ist für die Veröffentlichung der zuvor erstellten Webseite mit GitLab Pages verantwortlich. Daher erhielt dieser Job den Namen `pages`. Außerdem ist es notwendig, dass alle für die Webseite benötigten Dateien im Ordner `public` zu finden sind, da GitLab anhand dieses Ordners die Webseite erstellt. Auf Grund dessen mussten die Artefakte aus den vorherigen Stages, die sich nicht, wie die verwendeten HTML- und JavaScript-Dateien, bereits im Ordner `public` befanden, dorthin verschoben werden, was in Code-Beispiel 17 zu sehen ist. Zum Schluss musste auch hier der Ordner `public` als Artefakt gespeichert werden, damit GitLab im Anschluss an diesen Job die Webseite mit Hilfe des Ordners veröffentlichen kann.

```
pages:
    stage: deploy
    script:
        - mv results.json public/results.json
        - mv pin_logs/ public/pin_logs/
    artifacts:
        paths:
            - public/
        expire_in: "1 hour"
```

Code-Beispiel 17: Veröffentlichung der Webseite in der CI/CD-Pipeline

Zuletzt soll ein Job in der Stage `result` über einen möglichen Abbruch der Pipeline entscheiden, um eine automatisierte Benachrichtigung über gefundene Unterschiede per Email zu ermöglichen. Dazu wurde ausschließlich die in der Stage `compare` erstellte Datei, die den Error-Code des JavaScript-Programms beinhaltet, benötigt. Auf Grund dessen wurden in diesem Job `dependencies` verwendet, welche dafür sorgen, dass nicht alle Artefakte der vorherigen Stages, sondern in diesem Fall, wie es in Code-Beispiel 18 definiert wurde, nur die Artefakte des Jobs `compare_pinning` geladen werden. Die Datei, die den Error-Code enthält, wird anschließend ausgelesen und die Pipeline wird abgebrochen, falls der ausgelesene Error-Code nicht 0 ist, also falls das Programm in der Stage `compare` mindestens einen Unterschied zwischen den beiden Varianten des Pinnings gefunden hat. Bei einem Abbruch der Pipeline sendet GitLab dann per Email eine Benachrichtung über den Abbruch, wodurch implizit auf vorhandene Unterschiede hingewiesen wird. [14]

```
check:
    stage: result
    dependencies:
        - compare_pinning
    script:
        - exit_code=$(cat exit_code.txt)
        - if [ "$exit_code" -ne 0 ]; then
              echo "Errors occurred during the comparison.";
              exit 1;
          fi
```

Code-Beispiel 18: Entscheidung über einen möglichen Abbruch der CI/CD-Pipeline

5. Zusammenfassung und Ausblick

Im Rahmen dieser Seminararbeit wurde in GitLab mittels CI/CD ein allgemeiner Ansatz für die Automatisierung der Verifikation von Slurm Prozess Pinning entwickelt, der das Pinning auf einem HPC-System automatisiert mit dem simulierten Pinning des Pinning-Webtools vergleicht. Dazu wird das auf dem HPC-System verwendete Pinning zunächst ausgelesen, im Anschluss mit dem simulierten Pinning verglichen und die Ergebnisse des Vergleichs werden zum Schluss auf einer internen Webseite übersichtlich dargestellt. Zudem werden die Entwickler des Tools automatisch per Email informiert, falls beim Vergleich Unterschiede aufgefallen sind. In dieser Arbeit wurde der entwickelte Ansatz zunächst auf dem HPC-System JUSUF angewendet.

Durch diesen Ansatz ist es nun möglich Unterschiede zwischen dem Pinning auf einem HPC-System und dem simulierten Pinning im Pinning-Webtool frühzeitig zu erkennen und zu beheben, wodurch das Pinning-Webtool zuverlässiger wird. Ein weiterer möglicher Einsatzbereich ist die Nutzung dieser Pipeline, um beispielsweise neue Versionen des Ressourcenmanagers Slurm auf potenzielle Änderungen im Pinning-Verhalten zu überprüfen. Dazu könnte der entwickelte Ansatz auf kleinen Testsystemen, die abgesehen von der Knotenzahl dieselbe Hardware wie die großen Cluster enthalten, angewendet werden, da diese unter anderem dazu dienen geplante Updates zu testen, bevor sie auf die großen Cluster ausgerollt werden.

Zukünftig soll die entwickelte Pipeline um weitere HPC-Systeme und zusätzliche Vergleichsmöglichkeiten, wie etwa den Vergleich mit dem ausgelesenen Pinning aus vorherigen Pipeline-Durchläufen oder der Ausgabe des auf den HPC-Systemen verfügbaren Befehls `jsc-get-affinity`, erweitert werden. Zusätzlich soll der entwickelte Ansatz auch für eine Veröffentlichung vorbereitet werden.

A. Anhang

```
#include "mpi.h"
#include <omp.h>
#include <iostream>
#include <iomanip>
#include <unistd.h>
using namespace std;

int main(int argc, char *argv[]) {
    int size, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int thread = 0, np = 1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);

    usleep(50000);
    for (int i=0; i<size; i++){
        if(i==rank){
            #pragma omp parallel default(shared) private(thread, np)
            {
                np = omp_get_num_threads();
                thread = omp_get_thread_num();
                #pragma omp critical
                    cout << setw(6) << rank << setw(6) << thread << setw(6)
                         << sched_getcpu() << "  " << processor_name << "  "
                         << argv[0] << "\n" << flush;
            }
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Code-Beispiel 19: C++-Programm, welches das Pinning ausliest

Literatur

[1] „Hochleistungsrechnen | Speichern und Rechnen | Themen | Forschungsdaten und Forschungsdatenmanagement," besucht am 24. Sep. 2024. Adresse: https://forschungsdaten.info/themen/speichern-und-rechnen/hochleistungsrechnen/.

[2] „Pinning-Process-Simulation," besucht am 14. Nov. 2024. Adresse: https://apps.fz-juelich.de/jsc/llview/pinning/.

[3] „NVIDIA: What is HPC?" NVIDIA, besucht am 7. Nov. 2024. Adresse: https://www.nvidia.com/en-us/glossary/high-performance-computing/.

[4] „NUMA - HPC Wiki," besucht am 4. Nov. 2024. Adresse: https://hpc-wiki.info/hpc/NUMA.

[5] „Processor Affinity — JUSUF user documentation documentation," besucht am 4. Nov. 2024. Adresse: https://apps.fz-juelich.de/jsc/hps/jusuf/affinity.html.

[6] „Profile," besucht am 7. Nov. 2024. Adresse: https://www.fz-juelich.de/en/ias/jsc/about-us/profile.

[7] „Slurm Workload Manager - Overview," besucht am 24. Sep. 2024. Adresse: https://slurm.schedmd.com/overview.html.

[8] „Batch system — JUWELS user documentation documentation," besucht am 4. Nov. 2024. Adresse: https://apps.fz-juelich.de/jsc/hps/juwels/batchsystem.html.

[9] „Slurm Workload Manager - srun," besucht am 14. Nov. 2024. Adresse: https://slurm.schedmd.com/srun.html.

[10] „Processor Affinity — JUWELS user documentation documentation," besucht am 24. Sep. 2024. Adresse: https://apps.fz-juelich.de/jsc/hps/juwels/affinity.html.

[11] „Process distribution, affinity and binding — TGCC public documentation," besucht am 4. Nov. 2024. Adresse: https://www-hpc.cea.fr/tgcc-public/en/html/toc/fulldoc/Process_distribution_affinity_binding.html#process-and-thread-affinity.

[12] J. Wellmann, „Entwicklung einer webbasierten Visualisierung des Prozess-Pinnings auf HPC-Systemen," Course work, FH Aachen, 2020, 23 p. Adresse: https://juser.fz-juelich.de/record/893380.

[13] „Was ist eine CI/CD-Pipeline?" Besucht am 4. Nov. 2024. Adresse: https://about.gitlab.com/de-de/topics/ci-cd/cicd-pipeline/.

[14] „CI/CD YAML syntax reference | GitLab," besucht am 4. Nov. 2024. Adresse: https://docs.gitlab.com/ee/ci/yaml/.

[15] „GitLab runner | GitLab," besucht am 4. Nov. 2024. Adresse: https://docs.gitlab.com/runner/.

[16] „Docker Runner :: JSC GitLab documentation," besucht am 4. Nov. 2024. Adresse: https://gitlab.pages.jsc.fz-juelich.de/sharedrunner/docker/.

[17] „Job artifacts | GitLab,“ besucht am 4. Nov. 2024. Adresse: https://docs.gitlab.com/ee/ci/jobs/job_artifacts.html.

[18] „GitLab pages | GitLab,“ besucht am 4. Nov. 2024. Adresse: https://docs.gitlab.com/ee/user/project/pages/.

[19] „Jacamar CI Runners — JUWELS user documentation documentation,“ besucht am 4. Nov. 2024. Adresse: https://apps.fz-juelich.de/jsc/hps/juwels/jacamar.html.

[20] „SGML – SELFHTML-Wiki,“ besucht am 24. Sep. 2024. Adresse: https://wiki.selfhtml.org/wiki/SGML.

[21] „HTML/Tutorials/Entstehung und Entwicklung – SELFHTML-Wiki,“ besucht am 24. Sep. 2024. Adresse: https://wiki.selfhtml.org/wiki/HTML/Tutorials/Entstehung_und_Entwicklung.

[22] „Introduction to HTML,“ besucht am 24. Sep. 2024. Adresse: https://www.w3schools.com/html/html_intro.asp.

[23] „JavaScript/Tutorials/Einstieg – SELFHTML-Wiki,“ besucht am 4. Nov. 2024. Adresse: https://wiki.selfhtml.org/wiki/JavaScript/Tutorials/Einstieg.

[24] „JavaScript/Tutorials/DOM/Was ist das DOM – SELFHTML-Wiki,“ besucht am 5. Nov. 2024. Adresse: https://wiki.selfhtml.org/wiki/JavaScript/Tutorials/DOM/Was_ist_das_DOM.

[25] „Using the document object model - web APIs | MDN,“ besucht am 24. Sep. 2024. Adresse: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Using_the_Document_Object_Model.

[26] „Node.js — introduction to node.js,“ besucht am 18. Nov. 2024. Adresse: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs.

[27] „Node.js — about node.js®,“ besucht am 4. Nov. 2024. Adresse: https://nodejs.org/en/about.

[28] T. Breuer, J. Wellmann, F. Souza Mendes Guimarães, C. Himmels und S. Luehrs, *JUBE*, Version REL-2.7.1, Mai 2024. DOI: 10.5281/zenodo.11394333. Adresse: https://doi.org/10.5281/zenodo.11394333.

[29] „GitHub - FZJ-JSC/JUBE at REL-2.7.1,“ GitHub, besucht am 14. Nov. 2024. Adresse: https://github.com/FZJ-JSC/JUBE.

[30] „JUBE Benchmarking Environment,“ besucht am 14. Nov. 2024. Adresse: https://www.fz-juelich.de/en/ias/jsc/services/user-support/software-tools/jube.

[31] „Glossary — JUBE 2.7.1 documentation,“ besucht am 14. Nov. 2024. Adresse: https://apps.fz-juelich.de/jsc/jube/docu/glossar.html.

[32] „JUBE/bin/jube-autorun at master · FZJ-JSC/JUBE · GitHub,“ besucht am 15. Nov. 2024. Adresse: https://github.com/FZJ-JSC/JUBE/blob/master/bin/jube-autorun.