Fachhochschule Aachen, Campus Jülich

Fachbereich Medizintechnik und Technomathematik
im Studiengang Angewandte Mathematik und Informatik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Bachelorarbeit

Entwicklung einer automatisierten Testumgebung zur Analyse von Node-Sharing-Szenarien

von
Carina Himmels (Matrikelnummer: 3566808)

Jülich, den 21. August 2025

Diese Arbeit wurde betreut von:

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

2. Prüfer: M. Sc. Thomas Breuer

Firma: Forschungszentrum Jülich GmbH

Institut: Jülich Supercomputing Centre

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Eidesstattliche Erklärung

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

Entwicklung einer automatisierten Testumgebung zur Analyse von Node-Sharing-Szenarien

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 Bachelorarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Jülich, den 21. August 2025

Unterschrift der Studentin

Abstract

Das Jülich Supercomputing Centre (JSC) betreibt Höchstleistungsrechner, die darauf ausgelegt sind, Jobs auf vielen Rechenknoten gleichzeitig auszuführen. Da einzelne Jobs die Ressourcen eines Knotens – wie CPUs, GPUs oder Speicher – oft nicht vollständig auslasten, scheint Node-Sharing ein vielversprechender Ansatz zur Verringerung der Unterauslastung zu sein. Dabei teilen sich mehrere Anwendungen einen Rechenknoten, was jedoch zu gegenseitiger Konkurrenz um Ressourcen und Leistungseinbußen führen könnte.

Ziel dieser Arbeit war die Entwicklung einer Testumgebung zur Simulation und Analyse verschiedener Node-Sharing-Szenarien. Die Umgebung erlaubt es, typische Anwendungssituationen zu modellieren und Auswirkungen von Node-Sharing zu erkennen.

Erste exemplarische Analysen auf dem Höchstleistungsrechner JURECA demonstrieren die Funktionalität der Testumgebung und liefern erste Einblicke in die Auswirkungen von Node-Sharing auf die Leistung ausgewählter Anwendungen. Die entwickelte Umgebung bildet damit die Grundlage für weiterführende, systematische Untersuchungen von Node-Sharing auf Höchstleistungsrechnern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltext-Verzeichnis

Tabellenverzeichnis

Fachbegriffe und Akronyme

BLAS	Basic Linear Algebra Subroutines
CCD	Core Complex Die
CCX	Core Complex
CD	Continuous Delivery/Deployment
CG	Conjugate Gradients
CI	Continuous Integration
Cluster	Gruppe von vernetzten Computern
Commit	Momentaufnahme eines Projektes zu einem bestimmten Zeitpunkt
CPU	Central Processing Unit (Prozessor)
CSV	Comma-Separated Values
Daemon	Hintergrundprozess, der ohne Interaktion mit Benutzern Aufgaben erfüllt
FLOP/s	Floating Point Operations Per Second
GPU	Graphics Processing Unit (Grafikprozessor)
HPC	High Performance Computing
HPCG	High-Performance Conjugate Gradient (siehe Kapitel 3.4.2)
HPC-System	Höchstleistungsrechner (siehe Kapitel 1.2)
HPL	High-Performance LINPACK (siehe Kapitel 3.4.1)
I/O	Input/Output
Job	Ressourcenzuweisung, die einem Benutzer für eine bestimmte Zeit zugeteilt wird
Job-Schritt	Gruppe von Tasks innerhalb eines Jobs
JSC	Jülich Supercomputing Centre
JUBE	Jülich Benchmarking Environment (siehe Kapitel 3.1)
JURECA	Jülich Research on Exascale Cluster Architectures (siehe Kapitel 1.2.1)
Knoten	Parallel arbeitender Rechner in einem HPC-System (siehe Kapitel 1.2)
logische Kerne	Hardware-Threads innerhalb eines physischen Kerns
MPI	Message Passing Interface
Node-Sharing	Ausführen mehrerer Jobs auf einem Knoten (siehe Kapitel 2.2)
NUMA	Non-Uniform Memory Access (siehe Kapitel 1.2)
NVPL	NVIDIA Performance Libraries

OpenMP	Open Multi-Processing
physische Kerne	Kerne einer CPU
Pinning	Bindung eines Prozesses oder Threads an bestimmte Ressourcen (siehe Kapitel 2.1)
Plugin	Software-Erweiterung
Prozess	In sich abgeschlossene Ausführungseinheit, die aus einem oder mehreren Threads besteht
Slurm	Simple Linux Utility for Resource Management (siehe Kapitel 2)
Task	siehe Prozess
Thread	Teil eines Prozesses
URL	Uniform Resource Locator
Variationskoeffizient	Standardabweichung geteilt durch den Mittelwert
VSIPL	Vector Signal Image Processing Library
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

1 Einleitung

Auf Höchstleistungsrechnern kann die Zuweisung vollständiger Rechenknoten an Jobs zu ungenutzten Ressourcen führen. Um Node-Sharing, eine mögliche Lösung für die Unterauslastung, zu untersuchen, wurde in der vorliegenden Arbeit eine automatisierte Testumgebung entwickelt, mit der Node-Sharing-Szenarien in Zukunft analysiert werden können. Ziel ist es, anhand gemessener Metriken Aussagen darüber zu treffen, welche Auswirkungen Node-Sharing auf Anwendungen hat. [1]

Zu Beginn werden in diesem Kapitel die Motivation für die Themenwahl sowie grundlegende Informationen zu Höchstleistungsrechnern vorgestellt. Dazu zählt unter anderem der grundlegende Aufbau solcher Systeme. Darauf aufbauend wird in Kapitel 2 der Workload-Manager Slurm vorgestellt, der auf den Höchstleistungsrechnern des Jülich Supercomputing Centre (JSC) eingesetzt wird. Dabei werden auch zentrale theoretische Grundlagen, wie Prozess-Pinning und Node-Sharing, erläutert. Im weiteren Verlauf (Kapitel 3) werden die im Rahmen dieser Arbeit eingesetzten Softwarepakete und Anwendungen beschrieben, bevor in Kapitel 4 die Entwicklung der Testumgebung detailliert dargestellt wird. Bei der Implementierung wird dabei auf die zuvor vorgestellten Softwarepakete und Anwendungen zurückgegriffen. Nach der Implementierung der Testumgebung werden in Kapitel 5 verschiedene Node-Sharing-Szenarien analysiert, wobei die Testumgebung exemplarisch eingesetzt wird. Im letzten Kapitel, Kapitel 6, werden abschließend die wesentlichen Erkenntnisse zusammengefasst sowie mögliche, weiterführende Untersuchungen des Node-Sharings und Erweiterungen der entwickelten Testumgebung vorgestellt.

1.1 Motivation

High Performance Computing (HPC), also das Rechnen mit Höchstleistungsrechnern, hat sich in den letzten Jahren zu einer Schlüsseltechnologie in Wissenschaft und Wirtschaft entwickelt. Es ermöglicht das Modellieren, Simulieren, Visualisieren und Analysieren komplexer Phänomene sowie das Entwickeln von Prototypen. Das alles sind Aufgaben, die ohne HPC zu teuer, zu aufwendig oder schlicht unmöglich wären. Die Anwendungsbereiche reichen dabei von der globalen Klimamodellierung bis hin zur Erforschung neuer Energietechnologien. [2, 3]

Auch für das JSC hat HPC eine große Relevanz, da es modernste Höchstleistungsrechner, IT-Werkzeuge und auch das nötige Fachwissen für Forschende bereitstellt, die an über 200 Projekten in Deutschland und Europa beteiligt sind. [4]

Aktuell werden den einzelnen Berechnungen, den sogenannten Jobs, auf den HPC-Systemen des JSC ausschließlich die Ressourcen vollständiger Knoten zugewiesen. Diese Vorgehensweise wird auch von vielen anderen HPC-Centern genutzt, da dadurch Konflikte zwischen einzelnen Jobs vermieden werden können. Allerdings nimmt man dadurch eine Unterauslastung der HPC-Systeme in Kauf, da nicht alle Jobs alle Ressourcen eines Knotens nutzen. Eine Möglichkeit, der Unterauslastung entgegenzuwirken, bietet Node-Sharing. Beim Node-Sharing teilen sich mehrere Jobs einen Knoten, wodurch jedoch die Wahrscheinlichkeit steigt, dass Jobs sich gegenseitig behindern, da sie um Ressourcen konkurrieren, was zu Leistungseinbußen führen könnte. [1]

Ziel dieser Arbeit ist es, eine Testumgebung zu entwickeln, mit der sich verschiedene Node-Sharing-Szenarien simulieren und analysieren lassen. Mit Hilfe dieser Testumgebung soll es in Zukunft möglich sein, Aussagen darüber zu treffen, welche Auswirkungen Node-Sharing auf die Leistung von Anwendungen hat, um die Vor- und Nachteile von Node-Sharing abschätzen und fundierte Entscheidungen über einen möglichen Einsatz von Node-Sharing auf den Höchstleistungsrechnern des JSC treffen zu können. Zusätzlich sollen erste Anwendungen in die Testumgebung integriert werden, um bereits Jobs für verschiedene Node-Sharing-Szenarien auf dem Höchstleistungsrechner JURECA ausführen zu können, Metriken zu messen und eine exemplarische Analyse von Node-Sharing-Szenarien durchzuführen. Aufgrund des zeitlich eingeschränkten Rahmens dieser Arbeit erfolgt diese Analyse nur im begrenzten Umfang, um die Funktionalität der Testumgebung aufzuzeigen. Eine vertiefte und umfassende Untersuchung ist für zukünftige Arbeiten vorgesehen.

1.2 Höchstleistungsrechner

Höchstleistungsrechner zeichnen sich durch eine hohe Rechenleistung, umfangreiche Speicherkapazitäten und leistungsfähige Netzwerke aus, wodurch sie in der Lage sind, eine große Anzahl an Berechnungen parallel auszuführen. Verglichen mit handelsüblichen Rechnern lassen sich dadurch komplexe Berechnungen deutlich schneller durchführen, was das Lösen von Problemen ermöglicht, die mit herkömmlichen Rechenressourcen nicht zu bewältigen wären. Typische Anwendungsbereiche sind beispielsweise rechen- und datenintensive Klima-

simulationen. Darüber hinaus kommen Höchstleistungsrechner auch bei der Optimierung von Produkten oder Prozessen sowie bei der Analyse umfangreicher Datensätze aus Studien oder Experimenten zum Einsatz. [5]

Klassischerweise bestehen Höchstleistungsrechner aus zahlreichen Computern, den sogenannten Rechenknoten, welche parallel als ein einziges System zusammenarbeiten. Diese Knoten sind über ein Hochgeschwindigkeitsnetzwerk miteinander verbunden und verfügen jeweils über Arbeitsspeicher, lokalen Massenspeicher sowie eine oder mehrere Central Processing Units (CPUs). Manche Knoten sind zudem mit Graphics Processing Units (GPUs) ausgestattet. [5]

Als Speicherarchitektur wird im HPC-Bereich häufig Non-Uniform Memory Access (NUMA) verwendet. Jeder Rechenknoten bildet dabei ein eigenständiges NUMA-System, das aus mehreren Sockets mit jeweils einer CPU besteht. Jedem Socket ist dabei ein bestimmter Teil des Hauptspeichers als lokaler Speicher zugeordnet. Unter lokalem Speicher versteht man dabei einen Speicherbereich, auf den deutlich schneller zugegriffen werden kann, als auf andere Bereiche des Speichers. [6]

1.2.1 JURECA

Der für diese Arbeit genutzte Höchstleistungsrechner JURECA wird am JSC betrieben. Er setzt sich aus insgesamt 814 Knoten zusammen, von denen 780 jeweils zwei Sockets mit je einer AMD EPYC 7742 CPU beherbergen. [7, 8]

Eine solche CPU, deren schematischer Aufbau in Abbildung 1.1 dargestellt ist, ist aus mehreren Komponenten zusammengesetzt. Sie beinhaltet insgesamt acht sogenannte Core Complex Dies (CCDs) und einen I/O Die. Jeder CCD ist mit dem I/O Die verbunden und kann über diesen beispielsweise auf den Speicher zugreifen oder mit anderen CCDs kommunizieren. Darüber hinaus enthält jeder CCD zwei Core Complexes (CCXs), welche jeweils aus vier physischen Kernen bestehen, die sich einen 16 MB großen Level-3-Cache teilen. Jeder physische Kern verfügt dabei zudem über einen eigenen Level-1-Befehlscache mit 32 KB, einen Level-1-Datencache mit 32 KB und einen Level-2-Cache mit 512 KB. [9]

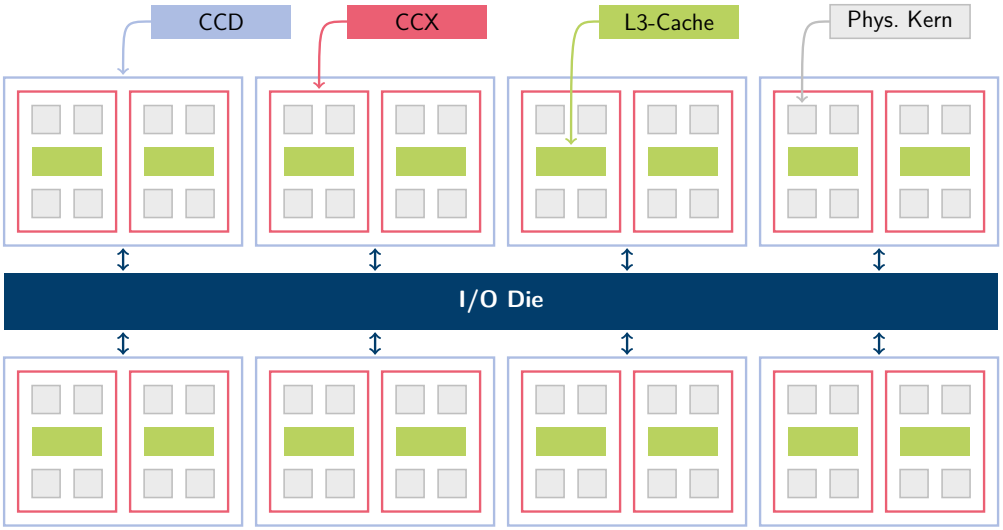

Abbildung 1.1: Aufbau der AMD EPYC 7742 CPU

Insgesamt enthält jede CPU somit 64 physische Kerne. Auf diesen können jedoch je zwei Threads parallel ausgeführt werden, sodass pro CPU 128 logische Kerne verfügbar sind. Dabei teilen sich die beiden logischen Kerne, die auf demselben physischen Kern ausgeführt werden, die zuvor erwähnten Level-1- und Level-2-Caches. In Abbildung 1.2, die einen gesamten Knoten von JURECA darstellt, sind die physischen Kerne als graue umrandete Kästchen dargestellt, während „X“ je einen logischen Kern repräsentiert. Um die Abbildung übersichtlich zu halten, wurden einige Kerne durch „...“ zusammengefasst. [9, 10]

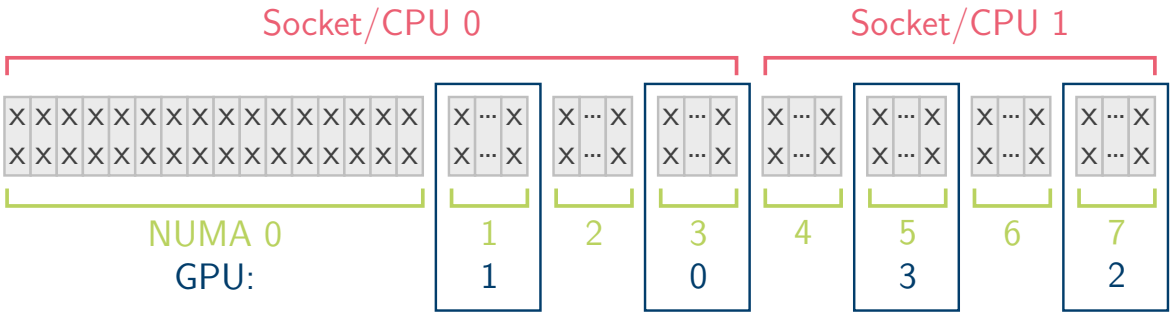

Abbildung 1.2: Aufbau von einem Knoten des HPC-Systems JURECA

Wie ebenfalls in Abbildung 1.2 zu sehen ist, kommt in JURECA eine NUMA-Speicherarchitektur zum Einsatz: Jeder Socket verfügt dabei über vier separate NUMA-Domänen, also vier lokale Speicherbereiche, mit jeweils 16 physischen Kernen. [10]

Des Weiteren enthalten 192 Knoten in JURECA neben den zwei AMD EPYC 7742 CPUs auch noch vier NVIDIA A100 GPUs, welche jeweils eine direkte Verbindung zu einer der NUMA-Domänen haben. Zu welchen NUMA-Domänen eine GPU eine direkte Verbindung hat, ist der Abbildung 1.2 zu entnehmen. [8, 10]

2 Slurm

Zur Zuweisung von Ressourcen an die Nutzer, zum Starten, Ausführen und Überwachen von Jobs sowie zur Verwaltung einer Warteschlange für anstehende Jobs wird auf den HPC-Systemen des JSC, und damit auch auf JURECA, Slurm eingesetzt. Slurm ist ein fehlertolerantes und hochskalierbares Cluster-Management- und Job-Scheduling-System, das weltweit auf Linux-Clustern unterschiedlicher Größe verwendet wird. [11, 12]

Insgesamt besteht es aus mehreren Programmen und Daemons, welche jeweils für unterschiedliche Bereiche zuständig sind. Der `slurmctld`-Daemon ist für die Überwachung aller Ressourcen und Jobs zuständig und läuft auf einem speziellen Verwaltungsknoten, wodurch im Falle von Hardwareausfällen die Verfügbarkeit dieses Daemons gewährleistet werden kann. Auf allen Rechenknoten läuft zusätzlich ein `slurmd`-Daemon zur Prozessverwaltung. Dieser wird auf den Systemen des JSC allerdings durch `psslurm` ersetzt – einem Plugin zu `psid`, dem Management-Daemon aus der Parastation Cluster Suite. Ansonsten stellt Slurm mehrere Benutzerprogramme wie `sbatch` oder `srun` zur Verfügung. [11, 12]

Mit Hilfe von `sbatch` kann ein Job, also die Zuweisung ausgewählter Ressourcen für einen bestimmten Zeitraum, angefordert werden. Dazu wird `sbatch` ein Batch-Skript, auch Job-Skript genannt, übergeben, in dem zu Beginn über die Direktive `#SBATCH` verschiedene Optionen angegeben werden können, welche unter anderem die benötigten Ressourcen spezifizieren. Zu diesen Optionen zählen beispielsweise `--nodes` oder `--threads-per-core`, mit denen die Anzahl der benötigten Knoten und die maximale Anzahl an verfügbaren logischen Kernen pro physischem Kern festgelegt werden kann. Ein Beispiel für ein Job-Skript befindet sich in Quelltext 2.1. In diesem Beispiel wird ein Job angefordert, der für maximal 5 Minuten einen Knoten zur Verfügung stellt, auf dem, im Fall von JURECA, alle logischen Kerne verwendet werden können. Typischerweise enthält ein solches Skript auch, wie in Zeile 6 des Beispiels, mindestens einen Aufruf des `srun`-Befehls, da mit diesem ein Job-Schritt, also eine Anwendung innerhalb eines Jobs, ausgeführt werden kann. In diesem Fall wird eine Anwendung ausgeführt, die insgesamt 10 Tasks mit je 2 Threads benötigt. [11]

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --threads-per-core=2
#SBATCH --time=00:05:00

srun --ntasks=10 --cpus-per-task=2 ./executable
```

Quelltext 2.1: Beispiel für ein Batch-Skript zur Anforderung eines Jobs

2.1 Prozess-Pinning

Prozess-Pinning ist ein Verfahren, das auf HPC-Systemen, und damit auch auf JURECA, von großer Bedeutung ist und über Slurm angepasst werden kann. Man versteht darunter das Binden von Prozessen an spezifische Kerne, wodurch Prozessmigration, also das Verschieben eines Prozesses auf andere Ressourcen, verhindert werden kann. Außerdem hat Prozess-Pinning einen großen Einfluss auf die Leistung von Anwendungen, da dadurch unter anderem Fernspeicherzugriffe reduziert werden können, welche beispielsweise bei NUMA-Architekturen auftreten können. [10, 13]

Slurm ermöglicht die Anpassung des Prozess-Pinnings über verschiedene Optionen des Benutzerprogramms `srun`, zu denen unter anderem `--cpu-bind` gehört. Mit `--cpu-bind` können verschiedene Bindungsvarianten, wie `mask_cpu` oder `map_cpu`, ausgewählt werden. Die Variante `mask_cpu` ermöglicht es beispielsweise, dass über hexadezimale Masken explizit festlegt wird, an welche logischen Kerne die Tasks einer Anwendung gebunden werden sollen. Anhand des in Quelltext 2.2 gezeigten Befehls, der insgesamt zwei Tasks erzeugt, wird die Funktionsweise von `mask_cpu` verdeutlicht: Die erste Task wird an die Kerne 0 und 1 gebunden, da $0\text{x}3 = 2^0 + 2^1$ ist. Nach demselben Prinzip wird die zweite Task an die Kerne 2 und 3 gebunden, da $0\text{x}\text{C} = 2^2 + 2^3$ ist. [10, 14]

```
srun -n 2 --cpu-bind=mask_cpu:0x3,0xC
```

Quelltext 2.2: Verwendung der Option `--cpu-bind=mask_cpu` des Slurm Befehls `srun` [10]

Die Variante `map_cpu` funktioniert ähnlich. Jedoch werden bei dieser Option statt hexadezimaler Masken die IDs der logischen Kerne angegeben. In dem Beispiel in Quelltext 2.3 wird daher die erste Task an den Kern mit der ID 1 gebunden, während die zweite Task an den Kern mit der ID 5 gebunden wird. Allerdings kann dabei für jede Task nur die ID eines einzigen Kerns angegeben werden, weshalb `map_cpu` nur verwendet werden kann, wenn jede erzeugte Task genau einen Thread beinhaltet. [10, 14]

```
srun -n 2 --cpu-bind=map_cpu:1,5
```

Quelltext 2.3: Verwendung der Option `--cpu-bind=map_cpu` des Slurm Befehls `srun` [10]

2.2 Node-Sharing

Node-Sharing ist eine Möglichkeit zur Verbesserung der Gesamtauslastung von HPC-Systemen durch das gleichzeitige Ausführen von mehreren Jobs auf einem Knoten. Dies trägt dazu bei, eine Unterauslastung der HPC-Systeme zu vermeiden, geht jedoch mit einer erhöhten Wahrscheinlichkeit einher, dass die Jobs sich gegenseitig behindern. Durch die gleichzeitige Ausführung mehrerer Jobs auf einem Knoten können die zur Verfügung stehenden Ressourcen stärker genutzt werden und es bleiben weniger Ressourcen ungenutzt. Gleichzeitig kann dies jedoch dazu führen, dass Jobs um die stark genutzten Ressourcen konkurrieren müssen. [1]

Da auf den HPC-Systemen des JSC kein Node-Sharing verwendet wird, um immer die größtmögliche Performanz gewährleisten zu können, muss dieses anderweitig simuliert werden. Dies ist durch den Einsatz von Slurm und Prozess-Pinning möglich. Über mehrere asynchrone `srun`-Aufrufe in einem Job-Skript können mehrere Anwendungen parallel auf den Knoten ausgeführt werden. Dabei kann über das Pinning explizit festgelegt werden, an welche Kerne die verschiedenen Anwendungen gebunden werden sollen, wodurch die Ressourcenzuweisung für Jobs auf demselben Knoten simuliert werden kann. Dieses explizite Pinning wird dabei rein zur gezielten Simulation von Node-Sharing verwendet und muss auf Systemen, auf denen tatsächlich Node-Sharing genutzt wird, nicht zwingend verwendet werden. Um über das Prozess-Pinning möglichst genau festlegen zu können, an welche Kerne die Anwendungen gebunden werden sollen, wird die `srun`-Option `--cpu-bind` genutzt, da diese es, wie in Kapitel 2.1 beschrieben, erlaubt, über die Werte `map_cpu` und `mask_cpu` das Pinning explizit anzugeben und damit das gewünschte Node-Sharing-Szenario festzulegen.

Zusätzlich muss zur Simulation von Node-Sharing die Option `--overcommit` genutzt werden, da die Nutzung der Option `--cpus-per-task`, die bei allen Anwendungen mit mehreren Threads pro Task benötigt wird, ansonsten dazu führt, dass es zu Fehlern mit `--cpu-bind=mask_cpu` bzw. `--cpu-bind=map_cpu` kommt. Die Fehler entstehen dadurch, dass `--cpus-per-task` implizit die Option `--exact` setzt. Durch diese Option steht Anwendungen nur noch die angeforderte Mindestanzahl an Kernen für das Pinning zur Verfügung und es können keine weiteren Kerne genutzt werden. Diese implizite Option kann durch `--overcommit` überschrieben werden, da `--overcommit` dafür sorgt, dass jeder Task alle logischen Kerne des Sockets zur Verfügung stehen. Außerdem sorgt `--exact` dafür, dass

parallele Job-Schritte, also parallele srun-Aufrufe, blockiert oder abgelehnt werden, solange die Option `--overlap` nicht angegeben wird. Daher wird auch diese Option zur Simulation von Node-Sharing benötigt, damit parallele srun-Aufrufe nicht blockiert werden, sondern tatsächlich parallel ausgeführt werden. [10, 14]

3 Verwendete Softwarepakete

Für die Implementierung der Testumgebung und zur Analyse der Node-Sharing-Szenarien wurden einige Softwarepakete und Anwendungen benötigt, welche im Folgenden kurz vorgestellt werden.

3.1 Die Benchmarking-Umgebung JUBE

JUBE [15], eine vom JSC entwickelte Benchmarking- und Workflow-Umgebung, wurde in dieser Arbeit verwendet, um Node-Sharing-Szenarien mit verschiedenen Anwendungen auszuführen und dabei Metriken zu ermitteln. Sie bietet einen skriptbasierten Ansatz zur Erstellung und Ausführung von Benchmark-Sets sowie zur Auswertung der Ergebnisse. [16]

3.1.1 Das JUBE-Skript

Die Umgebung unterstützt dazu zwei Eingabeformate: XML und YAML. Im Folgenden wird jedoch nur auf das XML-Format eingegangen, da beide dieselbe Funktionalität bieten. [17]

```
<jube>
    <benchmark name="..." outpath="...">
        <parameterset name="...">...</parameterset>
        <fileset name="...">...</fileset>
        <substituteset name="...">...</substituteset>
        <patternset name="...">...</patternset>
        <step name="...">...</step>
        <analyser name="...">...</analyser>
        <result>...</result>
        ...
    </benchmark>
</jube>
```

Quelltext 3.1: Allgemeiner Aufbau eines JUBE-Skriptes im XML-Format

Jedes Skript beginnt, wie in Quelltext 3.1 zu sehen ist, mit dem Element `<jube>`, in dem sich das `<benchmark>`-Element befindet. Dieses enthält alle weiteren Benchmark-Informationen. Außerdem wird über das Attribut `outpath` der relative Pfad zum Hauptverzeichnis angegeben, in dem für jeden Benchmark-Lauf ein neuer Ordner angelegt wird. [17, 18]

Erstellen von Parameterräumen

Um in JUBE einen Parameterraum festzulegen, können sogenannte Parametersets definiert werden, deren Struktur in Quelltext 3.2 anhand eines Beispiels dargestellt wird. Um innerhalb eines solchen Sets Parameter zu definieren, können Parameter über `<parameter>`-Elemente beschrieben werden. Der Wert innerhalb eines solchen Elementes entspricht dem Wert des Parameters. Statt eines einzigen Wertes können dort, wie in Quelltext 3.2, auch mehrere Werte über Kommata getrennt angegeben werden. JUBE spannt dann automatisch den vollständigen Parameterraum aus allen möglichen Kombinationen der angegebenen Parameter auf. Im Beispiel in Quelltext 3.2 würde dadurch ein Parameterraum mit insgesamt sechs verschiedenen Parameterkombinationen entstehen: „1 Zahl“, „2 Zahl“, „3 Zahl“, „1 Number“, „2 Number“ und „3 Number“. Dieser Parameterraum kann später von anderen Teilen des Skripts genutzt werden, um bestimmte Operationen automatisiert für jede dieser Kombinationen auszuführen. [17, 18]

```
1 <parameterset name="parameters">
2     <parameter name="number" type="int">1,2,3</parameter>
3     <parameter name="text">Zahl,Number</parameter>
4 </parameterset>
```

Quelltext 3.2: Beispiel für ein Parameterset in einem JUBE-Skript im XML-Format

Einbinden von Dateien

Außerdem kann während der Ausführung eines JUBE-Skriptes auf externe Dateien zugegriffen werden. Dazu können Befehle zum Kopieren und Verlinken in Filesets gespeichert werden. Innerhalb eines solchen Sets kann über `<copy>`, wie es in Quelltext 3.3 zu sehen ist, festgelegt werden, dass eine Datei ins entsprechende Arbeitsverzeichnis kopiert werden soll. Das Verlinken von Dateien funktioniert analog mit dem `<link>`-Element. Dadurch ist es beispielsweise möglich, den Quellcode oder Konfigurationsdateien in die jeweiligen von JUBE erzeugten Ordner zu kopieren. [18]

```
<fileset name="files">
    <copy>in.txt</copy>
</fileset>
```

Quelltext 3.3: Beispiel für ein Fileset in einem JUBE-Skript im XML-Format

Substitutionen in Dateien

Um Inhalte innerhalb von Dateien zu substituieren, stehen sogenannte Substitutesets zur Verfügung, in denen die eigentlichen Substitutionen über `<sub>`-Elemente definiert werden können. Zusätzlich muss über ein `<iofile>`-Element angegeben werden, in welcher Ein- und Ausgabedatei die Substitutionen durchgeführt werden soll. In dem Beispiel in Quelltext 3.4 wird festgelegt, dass der Inhalt der Datei `in.txt` in die Datei `out.txt` kopiert wird, wobei alle Stellen, an denen „#NUMBER#“ steht, durch den Wert des Parameters `number` ersetzt werden. Dadurch können beispielsweise Konfigurationsdateien dynamisch mit beliebigen, in JUBE definierten Parameterwerten modifiziert werden. [18]

```
<substituteset name="substitution">
    <iofile in="in.txt" out="out.txt" />
    <sub source="#NUMBER#" dest="$number" />
</substituteset>
```

Quelltext 3.4: Beispiel für ein Substituteset in einem JUBE-Skript im XML-Format

Ausführen von Operationen

Die zuvor vorgestellten Sets können in `<step>`-Elementen verwendet werden. In diesen wird festgelegt, auf welche Parameter-, File- oder Substitutesets ein Step Zugriff hat und welche Operationen ausgeführt werden sollen. Der Step in Quelltext 3.5 hat beispielsweise Zugriff auf das Parameterset `parameters`. Da der damit erzeugte Parameterraum insgesamt sechs Parameterkombinationen enthält, werden sechs Arbeitspakete erzeugt und damit die Operation `echo "$text $number"` sechsmal, also einmal pro Parameterkombination, ausgeführt. [17, 18]

```
<step name="execute_step">
    <use>parameters,files,substitution</use>
    <do>echo "$text $number"</do>
</step>
```

Quelltext 3.5: Beispiel für einen Step in einem JUBE-Skript im XML-Format

Extraktion von Daten

Zusätzlich zu den bisher vorgestellten Sets besteht in JUBE die Möglichkeit, sogenannte Patternsets zu definieren. Diese dienen dazu, beliebige Dateien innerhalb der JUBE-Ordnerstruktur, wie z.B. Ausgabedateien, zu parsen und Inhalte aus diesen zu extrahieren. Innerhalb eines solchen Sets können `<pattern>` definiert werden, die typischerweise einen regulären Ausdruck enthalten, mit dem Dateien durchsucht werden sollen. Das Pattern `number_pattern`, das in Quelltext 3.6 definiert wird, sucht demnach nach einer Textstelle, an der beispielsweise „Number 5“ steht. Dabei wird durch die Verwendung des in JUBE vordefinierten Patterns `jube_pat_int` nur die entsprechende Ganzzahl, also 5, extrahiert. [17, 18]

```
1 <patternset name="pattern">
2     <pattern name="number_pattern">Number $jube_pat_int</pattern>
3 </patternset>
```

Quelltext 3.6: Beispiel für ein Patternset in einem JUBE-Skript im XML-Format

Nach der Ausführung aller Steps können Dateien mit sogenannten Analysern untersucht werden, um relevante Daten aus bestimmten Dateien zu extrahieren. So wie jeder Step kann jeder Analyser über das `<use>`-Element auf zuvor definierte Sets, genauer gesagt Patternsets, zugreifen. Über `<analyse>`-Elemente wird dann festgelegt, welche Steps und Dateien analysiert werden sollen. In Quelltext 3.7 wird daher definiert, dass die Datei `out.txt` für jedes einzelne Arbeitspaket des Steps `execute_step` analysiert werden soll. [17, 18]

```
1 <analyser name="analyse_number">
2     <use>pattern</use>
3     <analyse step="execute_step">
4         <file>out.txt</file>
5     </analyse>
6 </analyser>
```

Quelltext 3.7: Beispiel für einen Analyser in einem JUBE-Skript im XML-Format

Erzeugen eines Ergebnisausgabeformats

Zuletzt können die Ergebnisse eines Benchmarks in verschiedenen Formaten ausgegeben werden. Dies geschieht über ein `<result>`-Element, wobei nun im `<use>`-Element nur Analyser verwendet werden können. Ein mögliches Ausgabeformat ist beispielsweise `<table>`, das in Quelltext 3.8 verwendet wurde und standardmäßig eine Ausgabe im CSV-Format erzeugt. [18]

```
<result>
    <use>analyse_number</use>
    <table name="result" sort="number">
        <column>number</column>
        <column>number_pattern</column>
    </table>
</result>
```

Quelltext 3.8: Beispiel für ein Result in einem JUBE-Skript im XML-Format

Tagging

Außerdem können innerhalb des gesamten JUBE-Skriptes Elemente durch die Verwendung von sogenannten Tags eingebunden oder ausgeschlossen werden. Dazu kann jedes Element – mit Ausnahme von `<jube>` – ein `tag`-Attribut enthalten, welches eine Liste von Tags enthält. Elemente mit einem solchen Attribut werden von JUBE bei der Ausführung des Skriptes nur berücksichtigt, wenn mindestens einer der angegebenen Tags aktiv ist. Ein Tag gilt dann als aktiv, wenn er beim Aufruf des Skriptes über die Kommandozeile explizit angegeben wurde. Auf diese Weise kann das Verhalten des JUBE-Skriptes zur Laufzeit dynamisch angepasst werden, um verschiedene Ausführungsszenarien gezielt abzubilden. In der zu implementierenden Textumgebung wurden diese Tags beispielsweise eingesetzt, um von außen steuern zu können, welche Node-Sharing-Szenarien mit JUBE ausgeführt werden sollen. [18, 19]

3.1.2 Kommandozeilenbefehle

Zur Ausführung eines solchen JUBE-Skriptes, genauer gesagt zur Ausführung der darin definierten Steps, steht der Befehl `jube run <file> -t <tags>` zur Verfügung, bei dem über die Option `-t` die verschiedenen Tags angegeben werden können, die aktiv sein sollen. Anschließend lässt sich mit dem Befehl `jube analyse <benchmark_dir>` der Analyse-Prozess, der über die Analyser definiert wurde, starten. Zuletzt kann mit `jube result <benchmark_dir>` das gewünschte Ausgabeformat generiert werden. Bei beiden zuvor genannten Befehlen bezeichnet `<benchmark_dir>` das Hauptverzeichnis für alle Benchmark-Läufe, das der Nutzer im JUBE-Skript frei wählen kann. [19]

Um die zuvor genannten Befehle automatisiert in der richtigen Reihenfolge auszuführen, steht ein Bash-Skript mit dem Namen `jube-autorun` zur Verfügung. Dieses ruft intern zunächst `jube run` auf und wartet darauf, dass alle Arbeitspakete ausgeführt wurden. Anschließend werden die relevanten Dateien mit `jube analyse` analysiert und danach wird

mit `jube result` eine entsprechende Ausgabe erzeugt. Über die Option `-r "..."` des `jube-autorun`-Skriptes können zusätzliche Argumente wie die Tags an den `jube run` Befehl übergeben werden. [20]

3.2 LLview

Um die mit Hilfe von JUBE extrahierten Metriken aus den ausgeführten Programmen graphisch darzustellen, wurde LLview [21] verwendet. LLview ist ein Softwarepaket zur Überwachung von Systemen, die von einem Ressourcenmanager und Scheduler-System wie Slurm gesteuert werden. Eine zentrale Rolle spielt dabei das sogenannte Job Reporting Modul: Es sammelt umfassende Informationen zu allen Jobs, die auf dem System laufen, und stellt diese übersichtlich zur Verfügung. Dazu sammelt LLview Daten aus verschiedenen Quellen im System und führt diese in einem Web-Portal zusammen. Das Modul bietet unter anderem tabellarische Übersichten aller Jobs mit Informationen zur Performanz sowie detaillierte Zeitverläufe wichtiger Metriken in Form interaktiver Graphen. [21, 22]

Zusätzlich zu diesen Funktionen für das Monitoring einzelner Jobs bietet LLview die Möglichkeit, die Ergebnisse von kontinuierlich wiederkehrenden Benchmarkläufen ebenfalls im Web-Portal bereitzustellen, was im Rahmen dieser Arbeit genutzt wurde. Genau wie bei den Job-Daten können auch die Ergebnisse der Benchmarks, wie in Abbildung 3.1 zu sehen, sowohl in Tabellen als auch in Graphen angezeigt werden. LLview erwartet dazu beispielsweise CSV-Dateien, in denen die relevanten Daten zu den Benchmarkläufen gespeichert sind, und kann diese in regelmäßigen Abständen automatisch aus einem GitLab-Repository oder einem beliebigen überwachten Ordner abrufen und in das Web-Portal integrieren. Wie die Daten dargestellt werden sollen, kann flexibel über YAML-Einstellungen konfiguriert werden. [23]

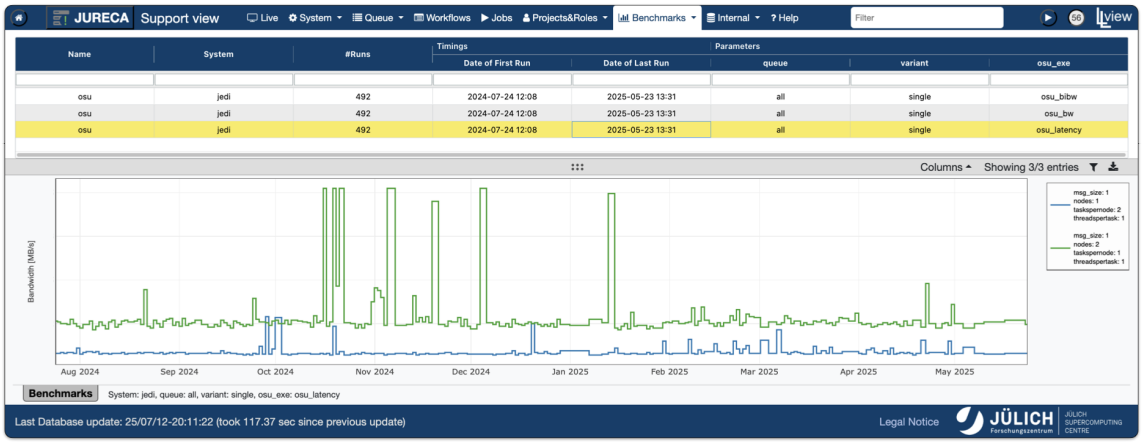

Abbildung 3.1: Continuous Benchmarking-Schnittstelle von LLview, die Ergebnisse des OSU-Benchmarks für das HPC-System JEDI anzeigt [23]

3.3 CI/CD

Da das gesamte Setup zur Ausführung von JUBE und zur Bereitstellung der Ergebnisse vollständig automatisiert und kontinuierlich ausführbar sein sollte, wurde zusätzlich eine CI/CD-Pipeline definiert, die das Starten von JUBE und das anschließende Bereitstellen der Ergebnisse automatisiert. Dadurch können die Ergebnisse zudem direkt in ein GitLab-Repository übertragen werden, was eine automatische Weiterverarbeitung durch LLview ermöglicht.

3.3.1 CI/CD in GitLab

Diese CI/CD-Pipeline wurde in einer JSC-internen GitLab-Instanz, einem beliebten Open-Source-Tool, definiert und ausgeführt. Dazu wurde eine YAML-Datei mit dem Namen `.gitlab-ci.yml` angelegt, in der die Pipeline über sogenannte Stages definiert wird. Jede Stage wiederum besteht aus mehreren sogenannten Jobs. [24, 25]

Die Stages werden über das Schlüsselwort `stages` definiert, wobei die Reihenfolge der Einträge auch die Ausführungsreihenfolge der verschiedenen Stages angibt. Im Beispiel in Quelltext 3.9 würden also zuerst alle Jobs aus der Stage `build` und dann alle Jobs aus der darauffolgenden Stage `test` ausgeführt werden. Dabei ist zu beachten, dass jede Stage aus mehreren Jobs bestehen kann, deren Definition im gezeigten Beispiel jedoch nicht enthalten ist. [25]

```
1 stages:
2   - build
3   - test
```

Quelltext 3.9: Verwendung des `stages`-Schlüsselwortes in der Datei `.gitlab-ci.yml`

Ein einzelner Job wird über verschiedene Schlüsselworte definiert und über einen Namen identifiziert. In dem Beispiel in Quelltext 3.10 hat der Job beispielsweise den Namen `test_job`. Über das Schlüsselwort `stage` wird ein Job einer Stage zugeordnet, in der dieser dann ausgeführt wird, und die auszuführenden Befehle werden über das `script`-Schlüsselwort angegeben. Der Job in Quelltext 3.10 läuft somit in der `test`-Stage und gibt „Testen...“ aus. [25]

```
1  test_job:
2    stage: test
3    tags:
4      - shell
5    script:
6      - echo "Testen..."
7    artifacts:
8      paths:
9        - out.txt
10     expire_in: 1 hour
```

Quelltext 3.10: Beispiel für die Definition eines Jobs in der Datei `.gitlab-ci.yml`

Damit ein Job tatsächlich ausgeführt werden kann, ist ein sogenannter Runner erforderlich. Man versteht darunter einen Prozess, der einen Job ausführt. Jeder Runner wird dazu mit der Anwendung GitLab Runner bei GitLab registriert, um unter anderem die Kommunikation zwischen GitLab und dem Rechner, auf dem die Anwendung GitLab Runner installiert ist, zu ermöglichen und sogenannte Tags festzulegen, über die ein Runner identifiziert werden kann. Um einem Job dann einen spezifischen Runner zuzuordnen, können über das Schlüsselwort `tags` die zuvor festgelegten Tags angegeben werden, die den entsprechenden Runner identifizieren. [25, 26]

Da es in bestimmten Fällen auch erforderlich sein kann, Dateien oder Verzeichnisse, die während der Ausführung eines Jobs entstehen oder verändert wurden, auch nach dem Job-Durchlauf zu speichern, können solche Dateien bzw. Verzeichnisse, wie in Zeile 7-10 des Quelltextes 3.10, über das `artifacts`-Schlüsselwort als Artefakt gespeichert werden. Artefakte sind Listen von Dateien und Verzeichnissen, die einem bestimmten Job zugeordnet werden und gespeichert bleiben. Sie ermöglichen es beispielsweise, Ergebnisse aus einer früheren Stage in späteren Stages weiterzuverwenden. [25]

3.3.2 Jacamar CI

Um im Rahmen einer CI/CD-Pipeline auf Höchstleistungsrechner zugreifen zu können, reichen die standardmäßig verwendeten GitLab Runner allerdings nicht aus. Aus diesem Grund kommen in dieser Arbeit zusätzlich Jacamar CI Runner zum Einsatz.

Jacamar CI ist ein Open-Source-Projekt, das eine Schnittstelle zwischen den GitLab CI/CD-Pipelines und Höchstleistungsrechnern bietet. Es verwendet einen Authentifizierungsmechanismus, der auf dem Abgleich der Benutzernamen zwischen GitLab und den Höchstleitungsrechnern basiert. Dadurch wird sichergestellt, dass nur berechtigte Nutzer CI/CD-Jobs auf den Höchstleistungsrechnern ausführen können. [27]

Im GitLab des JSC sind die Jacamar CI Runner für die Höchstleistungsrechner des JSC bereits als gemeinsam genutzte Runner eingerichtet. Auf jedem System sind 2 verschiedene Runner verfügbar: ein Shell-Runner, der Operationen auf den Login-Knoten des Systems ausführt, sowie ein Slurm-Runner, der Jobs direkt über Slurm anfordert. Die auf JURECA verfügbaren Runner sind in der Tabelle 3.1 aufgelistet. [27]

Runner	GitLab CI/CD Tags
Jacamar Slurm-Runner	jureca, jacamar, compute, slurm
Jacamar Shell-Runner	jureca, jacamar, login, shell

Tabelle 3.1: Verfügbare Jacamar CI Runner auf JURECA [27]

3.4 Zur Analyse verwendete Anwendungen

Zusätzlich wurden noch Anwendungen für die erste exemplarische Analyse von Node-Sharing-Szenarien benötigt, welche in die Testumgebung integriert werden konnten. Ausgewählt wurden dazu zwei etablierte synthetische Benchmarks, für die bereits je ein JUBE-Skript zur Kompilierung und Ausführung zur Verfügung stand, welches allerdings für die Verwendung innerhalb der Testumgebung modifiziert werden musste. Die beiden Benchmarks werden im Folgenden vorgestellt:

3.4.1 HPL

Die erste ausgewählte Anwendung ist das Softwarepaket HPL [28], welches in der Testumgebung auf den CPU-Kernen ausgeführt wird. Dieses generiert auf Computern mit verteiltem Speicher ein zufälliges dichtes lineares Gleichungssystem, löst es in 64-Bit-Gleitkommaarithmetik, überprüft die gefundene Lösung und misst die benötigte Zeit. Zur parallelen Berechnung nutzt HPL das Message Passing Interface (MPI). Zusätzlich werden entweder die Basic Linear Algebra Subroutines (BLAS) oder die Vector Signal Image Processing Library (VSIPL) benötigt, da diese portable Routinen für Operationen der linearen Algebra bereitstellen. [29]

HPL ist aktuell die am weitesten verbreitete Metrik für die Bewertung von HPC-Systemen und wird beispielsweise zur Erstellung der TOP500-Liste [30], die die 500 leistungsstärksten Höchstleistungsrechner vergleicht, genutzt. Besonders Anfang der 1990er-Jahre, als HPL bekannt wurde, stimmte das ermittelte Ranking zusätzlich sehr gut mit der tatsächlichen Leistung realer Anwendungen überein. Warum diese Übereinstimmung heute nicht mehr uneingeschränkt gegeben ist, wird in Kapitel 3.4.2 im Zusammenhang mit dem HPCG-Benchmark näher erläutert. [31]

Der Algorithmus, den HPL verwendet, läuft folgendermaßen ab: Zu Beginn generiert HPL ein zufälliges lineares Gleichungssystem der Ordnung N:

$$A\vec{x} = \vec{b}, \qquad A \in \mathbb{R}^{N \times N}, \vec{x}, \vec{b} \in \mathbb{R}^{N}$$

Die dabei erzeugte Matrix A ist eine dichte Matrix, deren Einträge zufällig aus dem Intervall $[-1, 1]$ gewählt werden. Nachdem das Gleichungssystem definiert ist, löst HPL es durch eine klassische LU-Zerlegung. Um die Berechnung parallel durchzuführen, werden die Daten blockzyklisch auf ein zweidimensionales Gitter von Prozessen verteilt, welches die Dimensionen P und Q hat. Die erweiterte Koeffizientenmatrix $[A, \vec{b}]$ wird dazu logisch in Blöcke mit einer festen Blockgröße von $NB \times NB$ aufgeteilt, welche auf das $P \times Q$-Gitter der Prozesse verteilt werden. Veranschaulicht wird die Verteilung der Daten in Abbildung 3.2, wobei die Abkürzungen `P0` bis `P5` für die Prozesse 0 bis 5 stehen. [29]

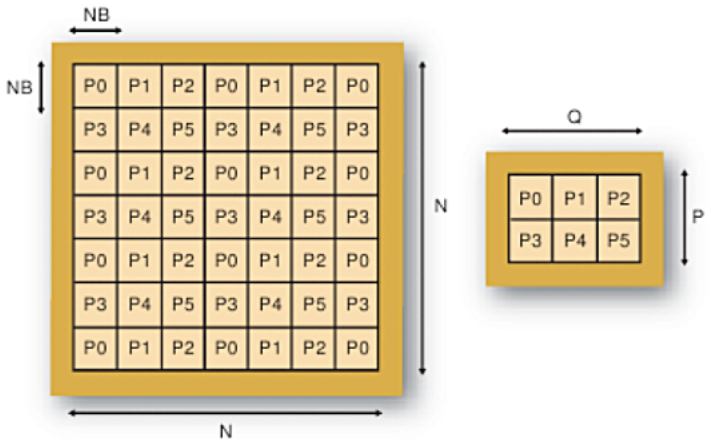

Abbildung 3.2: Verteilung der Daten bei HPL [32]

Die Ausgabe von HPL enthält zum einen eine Auflistung aller verwendeten Problemgrößen, wie N, NB, P oder Q, und zum anderen die Ergebnisgrößen des jeweiligen Laufes. Wie im Folgenden in Quelltext 3.11, einem Ausschnitt der Standardausgabe von HPL, zu sehen ist, wird sowohl die gemessene Laufzeit als auch die erreichten Floating Point Operations Per Second (FLOP/s) ausgegeben.

```
1  ================================================================================
2  T/V                N    NB     P     Q               Time                 Gflops
3  --------------------------------------------------------------------------------
4  WR01L2R4       82824   232     4     8             249.52             1.5180e+03
5  HPL_pdgesv() start time Tue Jul  8 12:32:21 2025
6
7  HPL_pdgesv() end time   Tue Jul  8 12:36:31 2025
```

Quelltext 3.11: Beispiel für die Ausgabe von HPL

In dieser Arbeit wurde für die Ausführung von HPL eine Konfiguration der Problemgrößen verwendet, welche bereits im JUBE-Skript für den HPL-Benchmark vordefiniert war. Die genutzten Werte für N, NB, P und Q sind Quelltext 3.11 zu entnehmen. Diese Konfiguration kann in Zukunft natürlich für mögliche weitere Untersuchungen gezielt angepasst werden.

3.4.2 HPCG

Die zweite ausgewählte Anwendung ist der HPCG-Benchmark [33], der in C++ implementiert ist und MPI und OpenMP unterstützt. Er wurde als Ergänzung zu HPL entwickelt, da HPL die Leistung von realen Anwendungen, die beispielsweise auf Speicherzugriffsmuster und Datenlokalität angewiesen sind, nicht mehr vollständig wiedergibt. Indem HPCG eine Vielzahl an Berechnungen und Datenzugriffsmustern simuliert, die in der wissenschaftlichen Datenverarbeitung üblich sind, bietet HPCG eine umfassendere und realitätsnähere Leistungsbewertung. [31, 34, 35]

Mit Hilfe des Conjugate Gradients (CG) Verfahrens und einem symmetrischen Gauß-Seidel-Vorkonditionierer löst der HPCG-Benchmark die Poisson-Differentialgleichung. Diese wird auf einem dreidimensionalen, rechteckigen Gitter mit festem Knotenabstand diskretisiert. Die globalen Dimensionen des betrachteten Raums sind $n_X P_X \times n_Y P_Y \times n_Z P_Z$, wobei $P_X \times P_Y \times P_Z$ ein dreidimensionales Gitter von Prozessen darstellt. Die Abmessungen eines einzelnen Teilbereichs, der jeweils einem Prozess zugeordnet wird, betragen $n_X \times n_Y \times n_Z$. Anschließend werden mehrere Sätze von CG-Iterationen mit derselben Anfangsschätzung durchgeführt, um die numerischen Ergebnisse am Ende durch Vergleichen auf ihre „Korrektheit“ überprüfen zu können. Das dabei verwendete vorkonditionierte CG-Verfahren ist in Abbildung 3.3 dargestellt. [31]

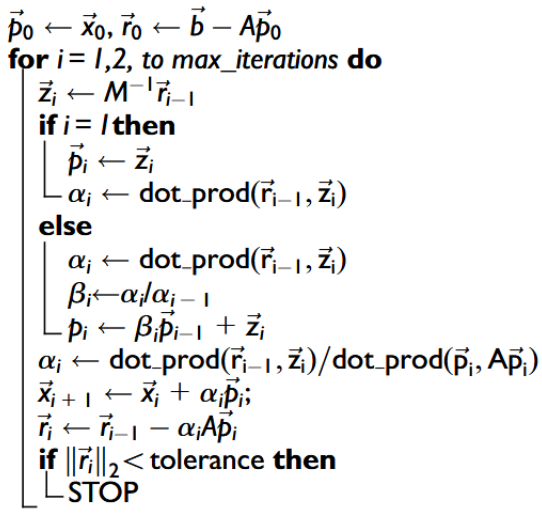

Abbildung 3.3: Von HPCG verwendetes vorkonditioniertes CG-Verfahren [31]

Nach einem Durchlauf erzeugt der HPCG-Benchmark eine ausführliche Ausgabe mit zahlreichen Werten. Diese enthält zunächst allgemeine Informationen zur verwendeten Konfiguration, wie die Anzahl der Prozesse oder die Gittergröße. Des Weiteren liefert sie Angaben zum Speicherverbrauch, zur Validierung der Ergebnisse und zur Gesamtperformanz. Beispielsweise werden Werte für die Bandbreite, die Rechenleistung in FLOP/s oder die Laufzeit angegeben. Abschließend folgt eine zusammenfassende Übersicht, für die in Quelltext 3.12 ein Beispiel zu sehen ist.

```
1 Final Summary::HPCG result is VALID with a GFLOP/s rating of=296.534
2 Final Summary::HPCG 2.4 rating for historical reasons is=299.968
3 Final Summary::Results are valid but execution time (sec) is=598.385
4 Final Summary::Official results execution time (sec) must be at least=1800
```

Quelltext 3.12: Beispiel für die Ausgabe von HPCG

Zusätzlich zum Standard-HPCG-Benchmark, der in der Testumgebung auf den CPU-Kernen ausgeführt wird, wurde in dieser Arbeit auch die Version von NVIDIA [36] verwendet. Durch den Einsatz der Mathematikbibliotheken cuSPARSE und NVIDIA Performance Libraries (NVPL) erzielt diese Version des HPCG-Benchmarks auf NVIDIA-CPUs und -GPUs eine optimale Leistung. Da JURECA über NVIDIA-GPUs verfügt, wird dieser Benchmark in der Testumgebung hauptsächlich auf den GPUs ausgeführt. [34]

Für beide Varianten wurden, wie bei HPL, Konfigurationen der Problemgrößen, die bereits im JUBE-Skript vordefiniert waren, als Grundlage genutzt. Im Rahmen dieser Arbeit wurden diese Konfigurationen an die verfügbaren Ressourcen und die spezifischen Anforderungen der Node-Sharing-Szenarien angepasst. Bei der Standardvariante des HPCG-Benchmarks

wurde die Gittergröße reduziert, da es zu Speicher- und Laufzeitproblemen kam. Statt der ursprünglichen Konfiguration wurde daher mit einer Gittergröße von $n_X = 128$ und $n_Y = n_Z = 64$ gearbeitet. Außerdem wurden abhängig von den in Kapitel 4.1 gewählten Node-Sharing-Szenarien 128 oder 64 Tasks eingesetzt. Die Verteilung der Tasks auf das dreidimensionale Prozessgitter erfolgte dabei durch HPCG selbst und ergab die Aufteilungen $P_X = 8, P_Y = P_Z = 4$ oder $P_X = P_Y = P_Z = 4$. Auch bei der NVIDIA-Variante des Benchmarks war eine Reduktion der Problemgrößen erforderlich, da die ursprünglich im JUBE-Skript definierten Werte in einigen Node-Sharing-Szenarien zu Speicherengpässen führten. In der angepassten Konfiguration wurde daher $n_X = 512, n_Y = 128, n_Z = 144$ verwendet. Die Ausführung erfolgte hierbei mit zwei Tasks, da einer Anwendung auf Grund der in Kapitel 4.1 beschriebenen Node-Sharing-Szenarien nur zwei GPUs zur Verfügung stehen. Die Verteilung auf das Prozessgitter wurde auch hier vom HPCG-Benchmark vorgenommen und ergab die Aufteilung $P_X = P_Y = 1, P_Z = 2$. Eine zukünftige Optimierung dieser Parameter ist möglich.

4 Entwicklung der Testumgebung

Um verschiedene Node-Sharing-Szenarien möglichst einfach analysieren zu können, wurde in dieser Arbeit eine Testumgebung entwickelt, mit deren Hilfe verschiedene Node-Sharing-Szenarien mit den zuvor vorgestellten Anwendungen auf einem HPC-System ausgeführt werden können. Diese Umgebung besteht wie zuvor beschrieben aus einer CI/CD-Pipeline, die in Abbildung 4.1 dargestellt ist. Sie führt mit JUBE Jobs für die verschiedenen Szenarien auf dem HPC-System JURECA aus und ermittelt Metriken, welche im Anschluss mit Hilfe von LLview und Python graphisch dargestellt werden.

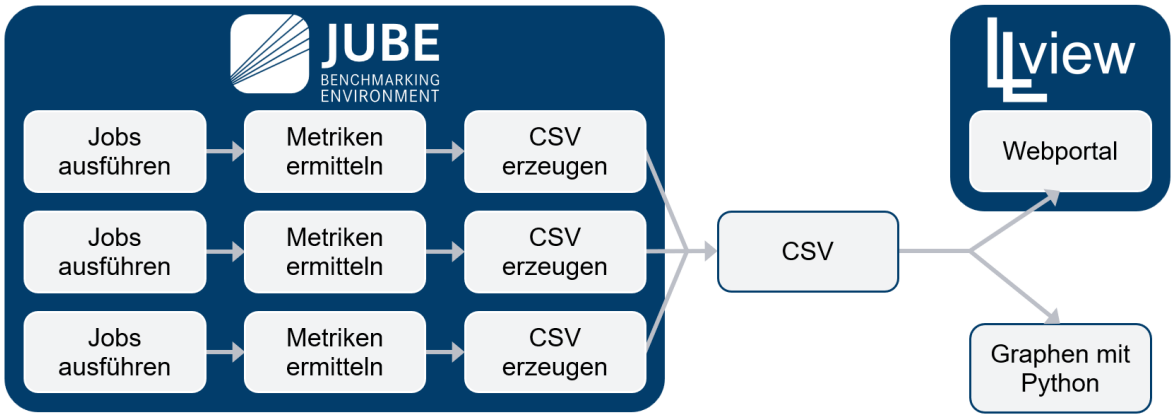

Abbildung 4.1: Schematische Darstellung der entwickelten CI/CD-Pipeline

4.1 Auswahl erster Node-Sharing-Szenarien

Bevor die Testumgebung implementiert werden konnte, mussten in dieser Arbeit zunächst geeignete Node-Sharing-Szenarien definiert werden. Um die Menge potenzieller Szenarien auf eine handhabbare Teilmenge zu reduzieren, die gleichzeitig die notwendigen technischen Grundlagen liefert, um später einfach erweitert werden zu können, wurde die Auswahl bewusst auf Szenarien beschränkt, die jeweils auf einem einzelnen Knoten ausgeführt werden und genau zwei Jobs umfassen. Gleichzeitig wurde bei der Auswahl der Szenarien darauf geachtet, verschiedene Aspekte der Ressourcennutzung, wie NUMA-Domänen, die Zuweisung an CPU-Kerne und die direkten GPU-Anbindungen, zu berücksichtigen. Auf Grundlage dieser Anforderungen wurden insgesamt sechs verschiedene Node-Sharing-Szenarien festgelegt. Jedes dieser Szenarien wird im Folgenden detailliert beschrieben und zusätzlich grafisch dargestellt. In den Grafiken stehen die gelb hinterlegten Kästchen dabei für die

logischen Kerne, die von der einen Anwendung genutzt werden, und die blau hinterlegten Kästchen entsprechend für die Kerne, die von der anderen Anwendung genutzt werden. Aus Darstellungsgründen wurden auch hier einige logische Kerne durch „...“ ersetzt.

Szenario „Separate CPUs“

Beim ersten Szenario, welches im folgenden „Separate CPUs“ genannt wird und in Abbildung 4.2 dargestellt ist, werden die beiden Anwendungen exklusiv einer einzelnen CPU zugewiesen. Dadurch nutzt die eine Anwendung alle Ressourcen der ersten CPU und die andere Anwendung alle Ressourcen der zweiten CPU. Somit kann jede Anwendung alle physischen Kerne einer CPU, die vier lokalen Speicherbereiche sowie die beiden direkten Verbindungen zu den GPUs nutzen, wobei ein Zugriff auf weitere GPUs systemseitig nicht ausgeschlossen ist. Dadurch kann untersucht werden, welche Auswirkungen Node-Sharing hat, wenn möglichst viele Ressourcen getrennt verwendet werden.

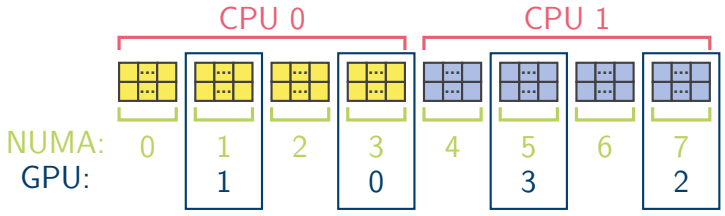

Abbildung 4.2: Schematische Darstellung des Node-Sharing-Szenarios „Separate CPUs“

Szenario „Separate NUMAs“

Das zweite Szenario teilt die NUMA-Domänen zyklisch zwischen den Anwendungen auf. Dadurch teilen sich die beiden Anwendungen nun die CPUs, haben aber dennoch eigene lokale Speicherbereiche, die sie sich nicht teilen müssen. Zusätzlich hat eine der beiden Anwendungen direkten Zugriff auf alle GPUs, während die andere Anwendung keinen direkten Zugriff auf GPUs hat. Dieses Szenario wird in Abbildung 4.3 abgebildet und wird im weiteren Verlauf als Szenario „Separate NUMAs“ bezeichnet. Es ermöglicht im Gegensatz zu Szenario „Separate CPUs“ die Untersuchung der Auswirkungen des Teilens einer CPU und damit der zugehörigen I/O-Dies auf die Performanz von Anwendungen.

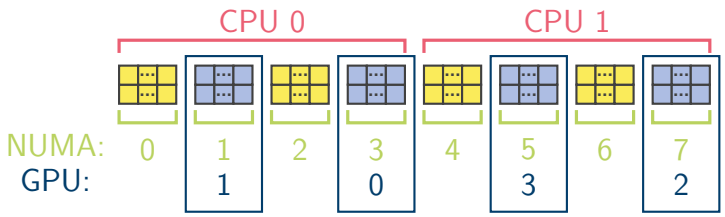

Abbildung 4.3: Schematische Darstellung des Node-Sharing-Szenarios „Separate NUMAs“

Szenario „Separate Cores“

Im dritten Fall „Separate Cores“, der in Abbildung 4.4 zu sehen ist, werden nicht nur die CPUs von den beiden Anwendungen geteilt, sondern auch die NUMA-Domänen, was eine weitere Steigerung der gemeinsamen Nutzung von Hardware-Ressourcen darstellt. Nur die CCDs und damit auch die physischen Kerne werden noch von nur einer Anwendung verwendet, da die eine Anwendung jeweils die ersten acht physischen Kerne und die andere Anwendung die letzten acht physischen Kerne einer NUMA-Domäne nutzt. Damit lässt sich nun zusätzlich untersuchen, welchen Einfluss eine gemeinsame Nutzung der lokalen Speicherbereiche auf die Performanz von Anwendungen hat.

Abbildung 4.4: Schematische Darstellung des Node-Sharing-Szenarios „Separate Cores“

Szenario „Shared Cores“

Das nächste Szenario „Shared Cores“ baut auf Szenario „Separate Cores“ auf, indem sich die Anwendungen nun zusätzlich die physischen Kerne der CPU teilen, indem jede Anwendung genau einen logischen Kern eines physischen Kerns nutzt, was in Abbildung 4.5 dargestellt ist. Damit teilen sich die Anwendungen nun auch die CCXs und die verschiedenen Caches, die in Kapitel 1.2.1 beschrieben wurden. Dadurch ist es nun möglich den Einfluss von geteilten physischen Kernen und Caches auf die Auswirkungen von Node-Sharing zu prüfen.

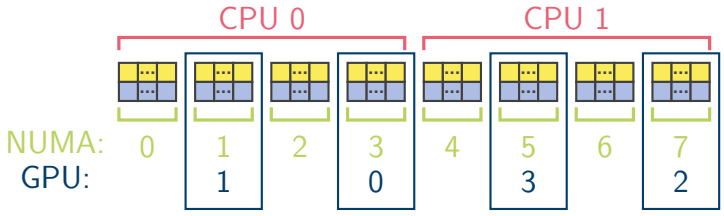

Abbildung 4.5: Schematische Darstellung des Node-Sharing-Szenarios „Shared Cores“

Szenario „Separate GPUs“

Die letzten beiden Szenarien sind hauptsächlich für GPU-lastige Anwendungen relevant, da hier im Gegensatz zu den bisherigen Szenarien nur die NUMA-Domänen mit einer direkten GPU-Anbindung genutzt werden. Die Untersuchung dieser Szenarien bietet einen Mehrwert, da in der Praxis häufig GPU-lastige Anwendungen gezielt nur die NUMA-Domänen mit direkter GPU-Anbindung nutzen, um von einer effizienteren CPU-GPU-Kommunikation zu

profitieren. Um vor allem die Kombination mehrerer GPU-Anwendungen im Kontext von Node-Sharing realitätsnah untersuchen zu können, ist es daher sinnvoll, diese typischen Nutzungsmuster nachzustellen und die NUMA-Domänen ohne direkte GPU-Anbindung in diesen Fällen unberücksichtigt zu lassen. [37]

In dem ersten GPU-Szenario nutzt jede Anwendung daher nur die NUMA-Domänen mit direkter GPU-Anbindung von je einer CPU. Da der direkte Zugang zu einer GPU nicht geteilt werden muss, wird das Szenario „Separate GPUs“ genannt. Abgebildet wird es in Abbildung 4.6. Außerdem ermöglicht das Szenario dadurch die Untersuchung, welchen Einfluss Node-Sharing auf die Performanz von GPU-lastigen Anwendungen hat, wenn alle Threads von einer direkten GPU-Anbindung profitieren.

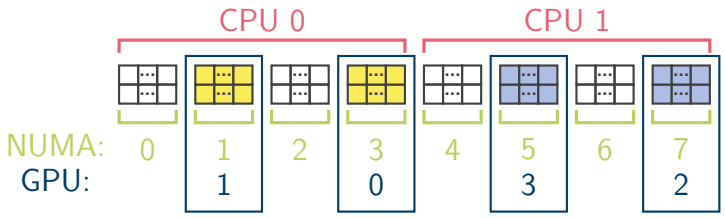

Abbildung 4.6: Schematische Darstellung des Node-Sharing-Szenarios „Separate GPUs“

Szenario „Shared GPUs“

Im zweiten GPU-Szenario teilen sich die Anwendungen im Gegensatz zu Szenario „Separate GPUs“ nun die NUMA-Domänen mit direkter GPU-Anbindung, weshalb durch dieses Szenario geprüft werden kann, welchen Einfluss Node-Sharing auf die Performanz von GPU-lastigen Anwendungen hat, wenn einige Threads keine direkte GPU-Anbindung nutzen können. Dieses Szenario wird „Shared GPUs“ genannt. Wie in Abbildung 4.7 zu sehen ist, nutzt dabei eine Anwendung, wie in Szenario „Separate Cores“, die ersten acht Kerne einer NUMA-Domäne und die andere Anwendung die letzten acht Kerne.

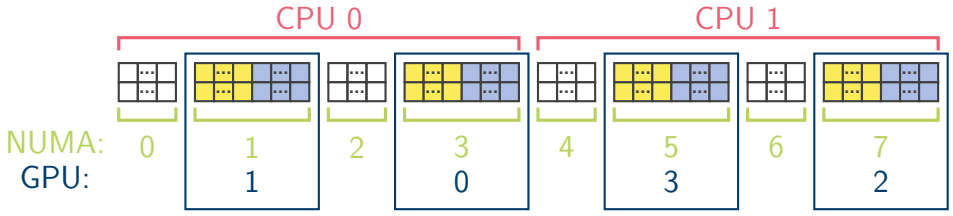

Abbildung 4.7: Schematische Darstellung des Node-Sharing-Szenarios „Shared GPUs“

Nutzung der GPUs

Die Abbildungen 4.2 bis 4.7 zeigen hauptsächlich die Nutzung der CPU-Ressourcen. Daher soll nun noch einmal kurz darauf eingegangen werden, wie die GPUs bei den einzelnen Szenarien verwendet werden, falls eine Anwendung die GPU-Ressourcen nutzt. So wie bei den CPU-Ressourcen verwendet jede Anwendung maximal die Hälfte der zur Verfügung stehenden GPUs. Da immer vier GPUs zur Verfügung stehen, bedeutet dies, dass jede Anwendung maximal zwei GPUs verwendet. In den Szenarien „Separate CPUs“ und „Separate GPUs“ verwendet jede Anwendung die GPUs, zu denen die genutzte CPU eine direkte Anbindung hat. In allen anderen Szenarien benutzen beide Anwendungen die Ressourcen von beiden CPUs und daher pro CPU je eine Anbindung zu einer GPU.

4.2 Ausführung der Node-Sharing-Szenarien

Um diese Szenarien mit verschiedenen Anwendungen auf einem HPC-System auszuführen, wurde ein JUBE-Skript geschrieben, welches Job-Skripte erzeugt und an den `sbatch`-Befehl übergibt, um Jobs anzufordern. Die Job-Skripte werden erzeugt, indem Platzhalter in einem Job-Skript-Template schrittweise ersetzt werden. Das bedeutet, dass die Platzhalter, die zum Ausführen der ersten Anwendung benötigt werden, zuerst ersetzt werden, dann die Platzhalter, die zum Ausführen der zweiten Anwendung benötigt werden, und zuletzt die Platzhalter, die den Job selber betreffen. Außerdem werden zu Beginn alle benötigten Anwendungen von dem JUBE-Skript einmal kompiliert.

Um auswählen zu können, für welche Anwendungen und Node-Sharing-Szenarien ein Job-Skript erzeugt werden soll, wurden JUBE-Tags eingeführt, die in der Tabelle 4.1 aufgelistet werden. Über diese kann genau ausgewählt werden, welche Anwendungen als „erste Anwendung“ und welche Anwendungen als „zweite Anwendung“ ausgeführt werden sollen, und auch die Szenarien können über diese Tags gewählt werden. Wenn beispielsweise die JUBE-Tags „`hpl0 hpcg_cpu1 no1 sep_cpu`“ beim Aufruf des JUBE-Skriptes angegeben werden, so erzeugt das Skript automatisch zwei Job-Skripte. Das erste führt HPL zusammen mit der Standardvariante von HPCG für das Node-Sharing-Szenario „Separate CPUs“ aus und das zweite führt nur HPL für dasselbe Node-Sharing-Szenario aus, sodass nur die Kerne der CPU 0 belegt werden.

	JUBE-Tags
Anwendung für den ersten `srun`-Aufruf	no0, hpl0, hpcg_cpu0, hpcg_gpu0
Anwendung für den zweiten `srun`-Aufruf	no1, hpl1, hpcg_cpu1, hpcg_gpu1
Node-Sharing-Szenario	sep_cpu, sep_numa, sep_core, share_core, sep_gpu, share_gpu

Tabelle 4.1: verfügbare Tags für das JUBE-Skript der Testumgebung

4.2.1 Erstellung des Job-Skript-Templates

Für die Implementierung wurde zunächst das Template für die Job-Skripte erstellt. Dieses startet mit verschiedenen `#SBATCH`-Direktiven, über welche beispielsweise die benötigten Ressourcen oder das Time-Limit für den Job festgelegt werden können. Die Werte für die einzelnen Optionen werden dabei noch nicht festgesetzt, sondern durch Platzhalter ersetzt. Danach folgen die Platzhalter für die beiden möglichen Anwendungen, die in Quelltext 4.1 zu sehen sind. Für die zweite Anwendung wird dabei jede „0“ in den Platzhaltern entsprechend durch eine „1“ ersetzt. Die Platzhalter in den Zeilen 1-3 können dabei beispielsweise genutzt werden, um benötigte Module zu laden oder Umgebungsvariablen zu exportieren. Die letzte Zeile der Platzhalter wiederum wird für den `srun`-Befehl benötigt, der durch das „&“ am Ende der Zeile asynchron ausgeführt wird. [38]

```
1 #ADDITIONAL_JOB_CONFIG0#
2 #ENV0#
3 #PREPROCESS0#
4
5 #MEASUREMENT0# #STARTER0# #ARGS_STARTER0# #EXECUTABLE0#
  ↪ #ARGS_EXECUTABLE0# &
```

Quelltext 4.1: Platzhalter im Job-Skript-Template für die erste Anwendung

Zuletzt wird auf alle asynchron ausgeführten Befehle, deren Prozess-IDs zuvor in der Variablen `pids` abgespeichert wurden, mit dem `wait`-Befehl gewartet. Der entsprechende Ausschnitt des Skriptes ist in Abbildung 4.2 angegeben. Falls es bei einem der `wait`-Befehle zu einem Exit-Code kommt, der nicht 0 ist, also falls es bei einem der srun-Befehle zu Fehlern kam, kann darauf über den Platzhalter `#FLAG_ERROR#` reagiert werden. Befehle, die ausgeführt werden sollen, wenn alle `srun`-Befehle fehlerfrei gelaufen sind, können über den Platzhalter `#FLAG#` angegeben werden. [38]

```
for pid in "${pids[@]}"; do
    wait $pid
    JUBE_ERR_CODE=$?
    if [ $JUBE_ERR_CODE -ne 0 ]; then
        #FLAG_ERROR#
        exit $JUBE_ERR_CODE
    fi
done
#FLAG#
```

Quelltext 4.2: Ausschnitt aus dem Job-Skript-Template zum Warten auf asynchrone Befehle

4.2.2 Vorbereitung der JUBE-Skripte der zu integrierenden Anwendungen

Da die Kompilierung einer Anwendung und das Ersetzen der Platzhalter im Job-Skript-Template voneinander getrennt ablaufen sollten, um eine wiederholte Kompilierung bei jeder Platzhalter-Ersetzung zu vermeiden, wurden die JUBE-Skripte aller zu integrierenden Anwendungen entsprechend angepasst. Dazu wurde in jedem Skript der Tag `compile` hinzugefügt. Falls dieser Tag bei der Ausführung der JUBE-Skripte angegeben wird, werden nun ausschließlich die für die Kompilierung relevanten Steps ausgeführt. Außerdem wurde ein neuer Step ergänzt, der die Ersetzung der Platzhalter im Job-Skript-Template übernimmt. Dazu wurde unter anderem auch je ein Substituteset ergänzt, das genau festlegt, welche Substitutionen vorgenommen werden sollen.

Da die JUBE-Skripte das Ersetzen der entsprechenden Platzhalter im Job-Skript-Template übernehmen, sind diese auch dafür verantwortlich, das Pinning entsprechend des Node-Sharing-Szenarios anzupassen. Daher werden in beiden JUBE-Skripten zusätzlich zu dem Tag `compile` auch noch die Tags für die Node-Sharing-Szenarien verwendet, die in Tabelle 4.1 zu sehen sind. Zwei weitere Tags, `exe0` und `exe1`, legen fest, ob für das Node-Sharing-Szenario das Pinning für die erste oder zweite Anwendung verwendet werden soll. Die verwendeten Pinning-Optionen entsprechen den in Kapitel 2.2 beschriebenen Optionen und wurden in Parametern gespeichert, um bei der Substitution darauf zugreifen zu können.

Schließlich wurden sämtliche Dateipfade, wie der zur ausführbaren Datei, angepasst, da sich durch die Trennung der Kompilierung und Ausführung der Anwendungen die Pfade zu den Dateien verändert haben. Bei all den zuvor genannten Änderungen wurde außerdem darauf geachtet, dass die JUBE-Skripte weiterhin auch getrennt von der Testumgebung fehlerfrei ausführbar sind.

4.2.3 Implementierung des JUBE-Skriptes für die Testumgebung

Danach wurde das JUBE-Skript für die Testumgebung implementiert. Dieses führt insgesamt die drei Steps aus, die als graue Kästchen in Abbildung 4.8 dargestellt sind. Wie dort zu erkennen ist, starten zwei Steps unabhängig voneinander und sind zum Einen für die Kompilierung der Anwendungen und zum Anderen für die Ersetzung der anwendungsspezifischen Platzhalter im Job-Skript-Template verantwortlich. Der letzte Step ist abhängig von den beiden und wird somit erst ausgeführt, wenn die beiden anderen Steps beendet wurden. Er ersetzt die letzten Job-spezifischen Platzhalter im Template und vervollständigt damit das Job-Skript. Dieses Job-Skript wird dann an den `sbatch`-Befehl übergeben, um den Job anzufordern und die Anwendungen auszuführen.

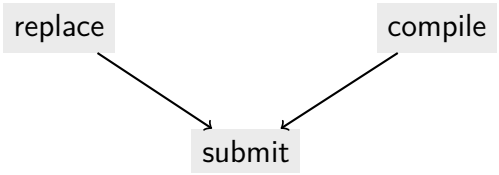

Abbildung 4.8: Ablauf der Steps im JUBE-Skript der Testumgebung

Für den `compile`-Step wird ein Fileset verwendet, welches das Verzeichnis verlinkt, in dem sich alle zu kompilierenden Anwendungen befinden. Innerhalb des Steps existiert für jede mögliche Anwendung ein `<do>`-Element, das über die entsprechenden Tags für die Anwendungen aktiviert wird. Jedes dieser `<do>`-Elemente führt, wie in Quelltext 4.3, das passende JUBE-Skript der Anwendung mit dem Tag `compile` aus, wodurch ausschließlich die Steps zur Kompilierung dieser Anwendung ausgeführt werden.

```
1 <do tag="hpl0|hpl1">
2     jube run apps/hpl.repo/hpl.xml -o hpl -t singlenode compile
3 </do>
```

Quelltext 4.3: `<do>`-Element im `compile`-Step aus dem JUBE-Skript der Testumgebung

Im `replace`-Step wird ebenfalls das zuvor genannte Fileset genutzt. Zusätzlich wird jedoch sowohl ein weiteres Fileset, welches das Job-Skript-Template in das aktuelle Arbeitsverzeichnis kopiert, als auch ein Parameterset verwendet, für das in Quelltext 4.4 ein Ausschnitt angegeben ist. Es enthält die Parameter `exe0`, `exe1` und `scenario`, in denen, abhängig von den verwendeten Tags, gespeichert wird, welche Anwendungen und welche Szenarien ausgeführt werden sollen.

```
<parameterset name="node-sharing-scenarios" duplicate="concat">
    <parameter name="exe0" tag="no0">-</parameter>
    <parameter name="exe0" tag="hpl0">hpl</parameter>
    <parameter name="exe0" tag="hpcg_cpu0">hpcg_cpu</parameter>
    <parameter name="exe0" tag="hpcg_gpu0">hpcg_gpu</parameter>
    ...
</parameterset>
```

Quelltext 4.4: Parameterset für den `replace`-Step aus dem JUBE-Skript der Testumgebung

In diesem Step existieren für jede Anwendung zwei `<do>`-Elemente: eines für den Fall, dass die Anwendung als „erste Anwendung“ verwendet wird, und eines für den Fall, dass sie als „zweite Anwendung“ verwendet wird. Innerhalb dieser `<do>`-Elemente wird das jeweilige JUBE-Skript der Anwendung aufgerufen. Wie in Quelltext 4.5 zu sehen, wird dieses Mal jedoch nicht der `compile`-Tag angegeben, sondern zum einen der Szenario-Tag für das Szenario der aktuellen Parameter-Kombination und zum anderen der passende `exe0`- bzw. `exe1`-Tag.

```
<do active="'${exe0}' == 'hpl'">
    jube run apps/hpl.repo/hpl.xml -o hpl0 -t singlenode ${scenario}
    ↪ exe0
</do>
```

Quelltext 4.5: Beispiel für ein `<do>`-Element im `replace`-Step aus dem JUBE-Skript der Testumgebung

Der letzte Step greift auf zwei weitere Parametersets zurück, die allgemeine Informationen, wie das Zeitlimit und die Menge der benötigten Ressourcen, enthalten. Außerdem wird ein Substituteset verwendet, das die verbleibenden Ersetzungen im Job-Skript vornimmt. Sobald das Job-Skript vollständig ist, wird es mit sbatch an Slurm übergeben.

4.3 Ermittlung und graphische Darstellung der Metriken

Nachdem die verschiedenen Node-Sharing-Szenarien über das JUBE-Skript ausgeführt werden konnten, wurden im nächsten Schritt die benötigten Ergebnisse und Metriken über `<analyser>`- und `<result>`-Elemente aus den Output- und Error-Dateien extrahiert und in LLview angezeigt. Dabei wurde sich auf die Metriken beschränkt, die jeweils in den finalen Ergebnis-Zusammenfassungen der Anwendungen angegeben werden. Das sind sowohl für HPL als auch für beide Varianten des HPCG-Benchmarks die Laufzeit in Sekunden und die

Rechenleistung in GFLOP/s. Insgesamt sollten in LLview daher folgende Daten angezeigt werden: die verwendeten Anwendungen, das verwendete Node-Sharing-Szenario sowie die gemessene Laufzeit und die Rechenleistung für alle verwendeten Anwendungen.

Zunächst wurde dazu die Schnittstelle zu LLview konfiguriert, indem über die YAML-Einstellungen festgelegt wurde, wo LLview die benötigten CSV-Dateien abrufen soll und wie die Daten dargestellt werden sollen. Dies wurde in Absprache mit einem LLview-Entwickler durchgeführt, der Zugriff auf die internen Dateien von LLview hat. Da sich die Kombinationen von Node-Sharing-Szenarien und Anwendungen bei mehrmaligem Ausführen der Testumgebung nicht ändern, sollten diese in einer Tabelle angezeigt werden. Weitere Standardspalten in der Tabelle sind das verwendete HPC-System und Zeitstempel des ersten und letzten Laufs. Sobald eine Tabellenzeile ausgewählt wird, sollten zusätzlich Graphen zu den beiden Metriken, also Laufzeit und Rechenleistung, angezeigt werden. Jeder Graph sollte dabei die Werte für beide Anwendungen und alle Läufe enthalten. Insgesamt ergibt sich durch diese Konfiguration die in Abbildung 4.9 abgebildete Webseite.

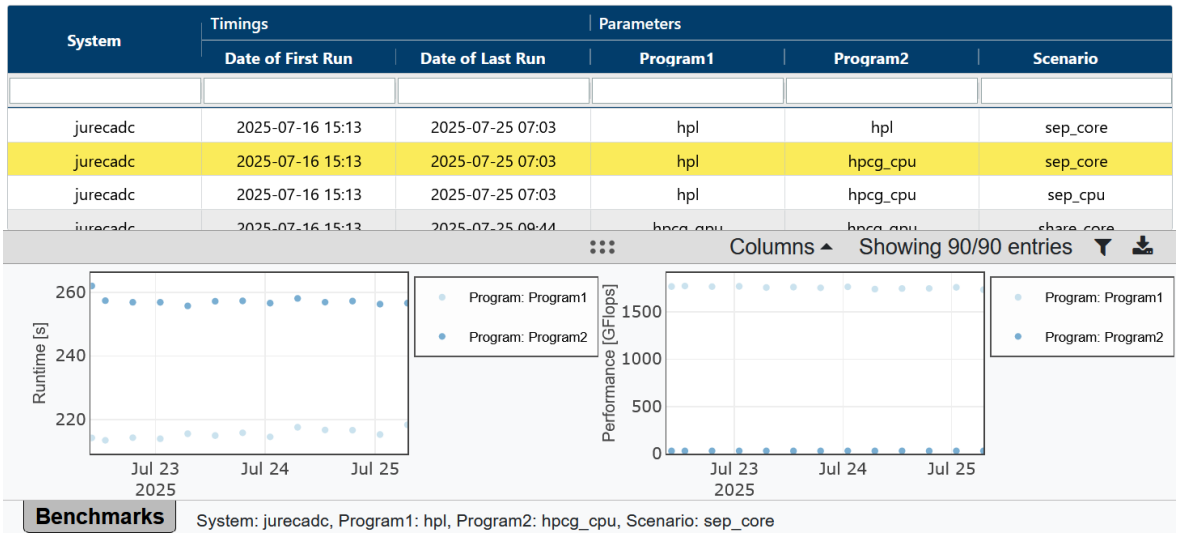

Abbildung 4.9: Continuous Benchmarking-Schnittstelle von LLview für die Testumgebung

Für diese Konfiguration wurde eine CSV-Datei benötigt, die die Spalten „system“, „timestamp“, „exe0“, „exe1“ und „scenario“ enthält und damit Informationen zu den verwendeten Anwendungen und dem verwendeten Szenario liefert. Außerdem wurden zwei Spalten für die Laufzeit und die Rechenleistung benötigt, die „time“ und „gflops“ heißen sollten. Um die Werte der Metriken der richtigen Anwendung zuzuordnen, wurde zusätzlich eine Spalte benötigt, die angibt, zu welcher Anwendung die jeweiligen Metriken gehören. Insgesamt wurde also für jede ausgeführte Anwendung der Jobs eine Zeile mit den folgenden Daten benötigt:

system, timestamp, exe0, exe1, scenario, result_exe, time, gflops

Erstellt wurde die CSV-Datei durch den Einsatz von JUBE. Dazu wurde dem JUBE-Skript der Testumgebung erst einmal ein `<result>`-Element hinzugefügt, welches eine Tabelle im CSV-Format mit den benötigten Spalten erzeugt. Zusätzlich zu den oben beschriebenen Spalten wurde noch eine weitere Spalte hinzugenommen, über die erkennbar ist, ob es bei der Ausführung eines Steps zu Fehlern gekommen ist. Da es in der Tabelle für jede Kombination von Szenario und Anwendungen zwei Zeilen geben sollte, je eine Zeile für die Metriken einer Anwendung, mussten hierbei zwei verschiedene Analyser verwendet werden. Der eine Analyser sollte alle Werte für die Zeile der ersten Anwendung ermitteln und der zweite Analyser dementsprechend alle Werte für die Zeile der zweiten Anwendung. Über Pattern wurde definiert, wie die benötigten Daten extrahiert werden können. [39]

4.4 Automatisierung mit GitLab CI/CD

Nachdem die Ausführung der Szenarien mit JUBE und die Erstellung der CSV-Datei für die Darstellung in LLview funktionierten, musste die erstellte CSV-Datei noch in ein GitLab Repository übertragen werden, damit LLview diese abrufen kann. Da das Übertragen der CSV-Datei nach jedem Durchlauf von JUBE ausgeführt werden muss und der gesamte Arbeitsablauf automatisiert ablaufen sollte, wurde eine CI/CD-Pipeline erstellt, die dies übernimmt. Diese Pipeline besteht insgesamt aus drei Stages: In der ersten Stage wird das JUBE-Skript je nach Konfiguration der Pipeline mehrfach ausgeführt, in der zweiten Stage werden die dabei erzeugten CSV-Dateien zu einer Datei zusammengefasst und bereinigt und in der letzten Stage wird diese CSV-Datei dann ins GitLab Repository der Testumgebung übertragen, damit LLview diese dort automatisch abrufen kann.

Die erste Stage enthält genau einen Job, der allerdings über die Schlüsselworte `parallel:matrix:` für verschiedene Kombinationen von JUBE-Tags ausgeführt wird. Ein einfaches Minimalbeispiel für die Verwendung von `parallel:matrix:` ist in folgendem Quelltext dargestellt:

```
1 test_job:
2   stage: test
3   script:
4     - echo "Running with ${VAR}"
5   parallel:
6     matrix:
7       - VAR: [var1, var2]
```

Quelltext 4.6: Beispiel für die Verwendung des Schlüsselworts `parallel:matrix:`

In diesem Fall würde der Job zweimal ausgeführt werden: einmal mit dem Umgebungswert `VAR=var1` und einmal mit `VAR=var2`. Im konkreten Kontext dieser Arbeit werden anstelle von `VAR` die in Tabelle 4.1 beschriebenen Tags verwendet. Die daraus resultierenden Teiljobs werden jeweils mit dem Shell-Runner auf JURECA ausgeführt. Jeder Teiljob lädt zunächst alle benötigten Module, wie beispielsweise JUBE, um im Anschluss das implementierte JUBE-Skript ausführen zu können. Zum Ausführen des JUBE-Skriptes wird dabei auf das in Kapitel 3.1.2 vorgestellte Bash-Skript `jube-autorun` zurückgegriffen, da dieses bereits automatisch die Ausführung, Analyse und Ausgabe der Ergebnisse übernimmt. Die erzeugte CSV-Datei wird dann als Artefakt gespeichert, damit in der zweiten Stage auf diese zugegriffen werden kann. Zuvor wird diese allerdings noch umbenannt, damit sich die Artefakte der Teiljobs nicht gegenseitig überschreiben. [25]

Die zweite Stage enthält ebenfalls genau einen Job. Dieser wird allerdings mit einem im GitLab des JSC verfügbaren Runner ausgeführt, der ein OpenSuse-basiertes System mit installierten Entwicklungspaketen zur Verfügung stellt. Damit bei einem Durchlauf der Pipeline nicht mehrere CSV-Dateien ins Repository übertragen werden, ist dieser Job dafür verantwortlich, die generierten CSV-Dateien der einzelnen Teiljobs aus der vorherigen Stage zu einer Datei zusammenzufassen und zu bereinigen. Dabei werden zum Beispiel alle Zeilen in der CSV-Datei entfernt, in denen es zu Fehlern bei der Ausführung des dazugehörigen JUBE-Steps kam. [40]

Die dadurch generierte CSV-Datei wird dann in der dritten Stage ins GitLab Repository übertragen, damit LLview diese automatisch abrufen kann. Zum Übertragen der Datei werden folgende Befehle ausgeführt:

```
1 script:
2 - git config user.email ${GITLAB_USER_EMAIL}
3 - git config user.name ${GITLAB_USER_NAME}
4 - git remote set-url origin https://ci_auth:${AUTOMATION_ACCESS_TOKEN}
   ↪  @${CI_PROJECT_URL#https://}.git
5 - git add results/result_${DATE}.csv
6 - git commit -m "added result of pipeline"
7 - git push origin -o ci.skip HEAD:$CI_COMMIT_REF_NAME
```

Quelltext 4.7: Ausschnitt aus dem Skript des letzten Jobs in der CI/CD-Pipeline der Testumgebung

Zunächst wird in Zeile 1 und 2 die Identität des Benutzers festgelegt, wobei die verwendeten Werte aus den CI/CD-Variablen stammen und den Namen bzw. die E-Mail-Adresse der Person enthalten, die die Pipeline gestartet hat. Anschließend wird die URL des Remote-Repositories angepasst. Dabei wird ein Project Access Token zur Authentifizierung verwendet.

Danach werden die generierten Dateien zur Staging-Area hinzugefügt, um im Anschluss einen Commit zu erstellen. Dieser wird zuletzt in Zeile 7 in das Remote-Repository übertragen. Dabei wird mit `ci.skip` sichergestellt, dass keine neue Pipeline erzeugt wird, damit nicht endlos neue Pipelines gestartet werden. [41–45]

Zusätzlich werden in einem weiteren Job in der letzten Stage Graphen für die Daten aus der generierten CSV-Datei erzeugt. Dazu wird ein Python-Skript, das die Pakete `matplotlib`, `pandas` und `numpy` nutzt, ausgeführt. Die Graphen ermöglichen es, die Laufzeiten und die Rechenleistung eines einzelnen Laufs miteinander zu vergleichen und werden beispielsweise in Kapitel 5.1 zur Analyse der Node-Sharing-Szenarien verwendet. Damit dies möglich ist, werden sie als Artefakt gespeichert.

5 Analyse der Node-Sharing-Szenarien

Nachdem die Testumgebung entwickelt wurde, konnte diese im Rahmen dieser Arbeit verwendet werden, um die in Kapitel 4.1 beschriebenen Node-Sharing-Szenarien mit verschiedenen Anwendungen auszuführen und dabei Metriken zu ermitteln, die nachfolgend untersucht werden. Diese Untersuchung ist allerdings, wie in Kapitel 1.1 bereits erwähnt, rein exemplarisch, da eine vertiefte und umfassende Untersuchung den Rahmen dieser Arbeit weit übersteigen würde.

Zuerst wurde die Testumgebung, genauer gesagt die implementierte CI/CD-Pipeline, genau einmal für alle möglichen Kombinationen von Anwendungen und Node-Sharing-Szenarien ausgeführt, um über diesen einzelnen Lauf erste Einblicke in die Auswirkungen von Node-Sharing auf die Laufzeit sowie die Rechenleistung zu erhalten. Im Anschluss wurden von den insgesamt 90 möglichen Kombinationen einige exemplarisch ausgewählt, für die die Auswirkungen von Node-Sharing mit Hilfe von LLview über die Zeit betrachtet wurden. Dazu wurde die CI/CD-Pipeline für die gewählte kleinere Menge von Kombinationen in regelmäßigen Abständen ausgeführt. Dadurch war es möglich, verschiedene Läufe miteinander zu vergleichen.

5.1 Erste Analyse durch einmalige Ausführung der Testumgebung

Nach der ersten einmaligen Nutzung der Testumgebung für alle möglichen Kombinationen von Anwendungen mit Node-Sharing-Szenarien, standen zur ersten Analyse für jede dieser Kombinationen sowohl die Laufzeit als auch die Rechenleistung beider Anwendungen zur Verfügung. Um tatsächlich Aussagen über die Auswirkungen von Node-Sharing treffen zu können, wurden die Metriken für jede Anwendung einzeln betrachtet. Dadurch war es möglich festzustellen, wie sich die Metriken verändern, wenn die Anwendung alleine läuft oder mit einer anderen Anwendung kombiniert wird.

5.1.1 Auswirkungen von Node-Sharing auf den HPL-Benchmark

Zuerst wurden die Metriken des HPL-Benchmarks betrachtet. Dabei zeigte sich, dass das Node-Sharing grundsätzlich einen Einfluss auf die Laufzeit von HPL hat, welcher jedoch je nach Szenario und Kombination mit anderen Anwendungen variiert. Das Säulendiagramm

in Abbildung 5.1 verdeutlicht diese Auswirkungen, da dort die Laufzeiten von HPL für die verschiedenen Kombinationen von Anwendungen und Node-Sharing-Szenarien angegeben werden.

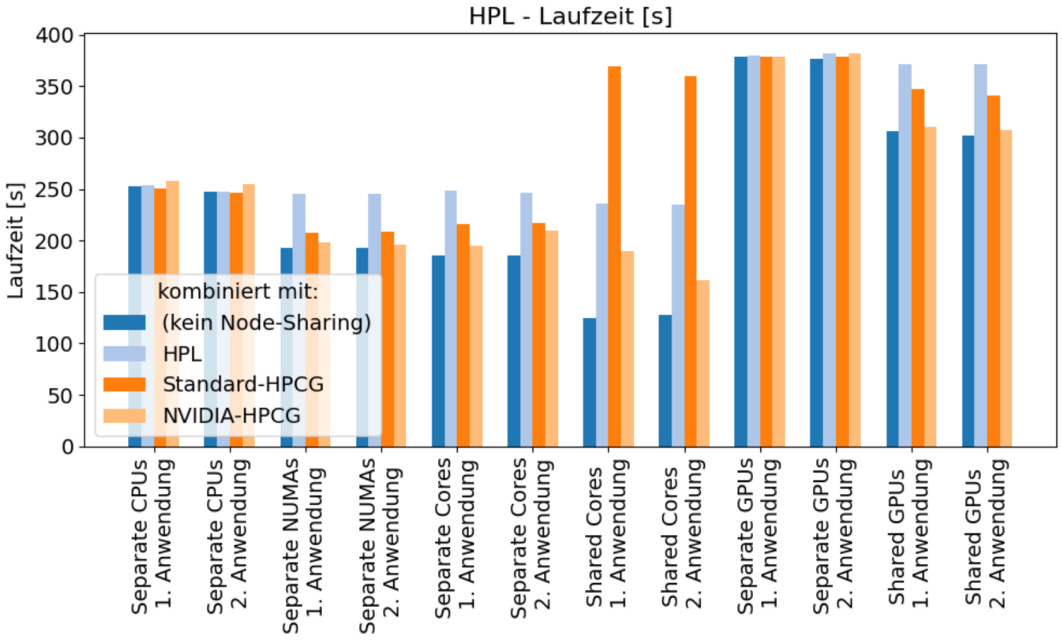

Abbildung 5.1: Laufzeiten des HPL-Benchmarks für verschiedene Szenarien

Im Allgemeinen ist festzustellen, dass die Laufzeit von HPL nahezu immer ansteigt, wenn Node-Sharing verwendet wird. Dabei beträgt die Erhöhung in manchen Fällen nur wenige Sekunden, in anderen wiederum mehrere Minuten. Ein besonderes Beispiel stellt das Szenario „Shared Cores“ dar, bei dem die Laufzeit von HPL um bis zu 244,88 Sekunden ansteigt, wenn Node-Sharing genutzt wird. Das entspricht einem Anstieg von 196,99%. Weitere Beobachtungen zeigen, dass die Kombination von HPL mit sich selbst tendenziell zu den längsten Laufzeiten führt. Dies liegt vermutlich daran, dass beide Instanzen in diesen Fällen exakt die gleichen Ressourcen beanspruchen und daher stark um Ressourcen konkurrieren müssen. Eine deutliche Ausnahme stellt hier das Szenario „Shared Cores“ dar, da HPL dort in Kombination mit dem Standard-HPCG-Benchmark deutlich längere Laufzeiten hat als in Kombination mit HPL selbst. Dieses Verhalten könnte auf die Konkurrenz um die CPU-Ressourcen zurückzuführen sein. Diese Vermutung lässt sich auch durch das Job Reporting Modul von LLview stützen: In Abbildung 5.2, einem Ausschnitt eines Job-Reports für einen dieser Fälle, ist zu erkennen, dass alle Kerne stark genutzt werden. Demnach beansprucht sowohl der Standard-HPCG-Benchmark, der in diesem Beispiel jeweils den „ersten“ logischen Kern eines physischen Kerns nutzt, als auch HPL, der die übrigen logischen Kerne nutzt, die CPU-Ressourcen stark.

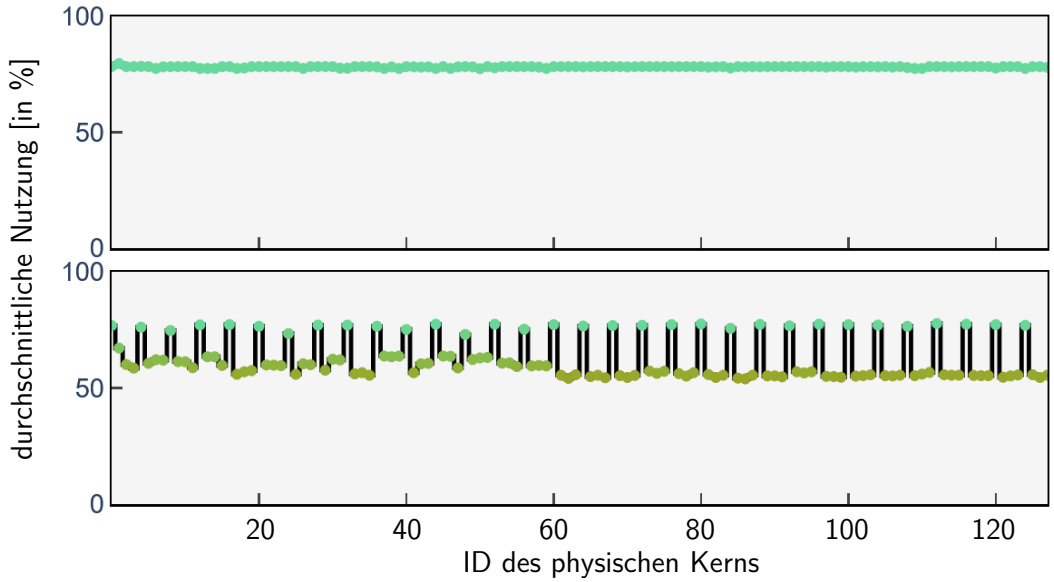

Abbildung 5.2: Nutzung des „ersten“ (oben) und des „zweiten“ (unten) logischen Kerns eines physischen Kerns beim Szenario „Shared Cores“ durch HPL und den Standard-HPCG-Benchmark

Genauso steigt die Laufzeit von HPL tendenziell am wenigsten an, wenn man als zweite Anwendung den NVIDIA-HPCG-Benchmark nutzt. Dies ist vermutlich darauf zurückzuführen, dass dieser Benchmark die CPU-Kerne nur geringfügig beansprucht und die Hauptlast auf den GPUs liegt, wodurch die CPU-Ressourcen für HPL weitgehend ungenutzt bleiben. Diese Vermutung lässt sich ebenfalls durch das Job Reporting Modul von LLview stützen: In Abbildung 5.3 ist zu erkennen, dass der NVIDIA-HPCG-Benchmark, der hauptsächlich die pink eingefärbten Kerne nutzt, die CPU-Kerne tatsächlich nur minimal auslastet.

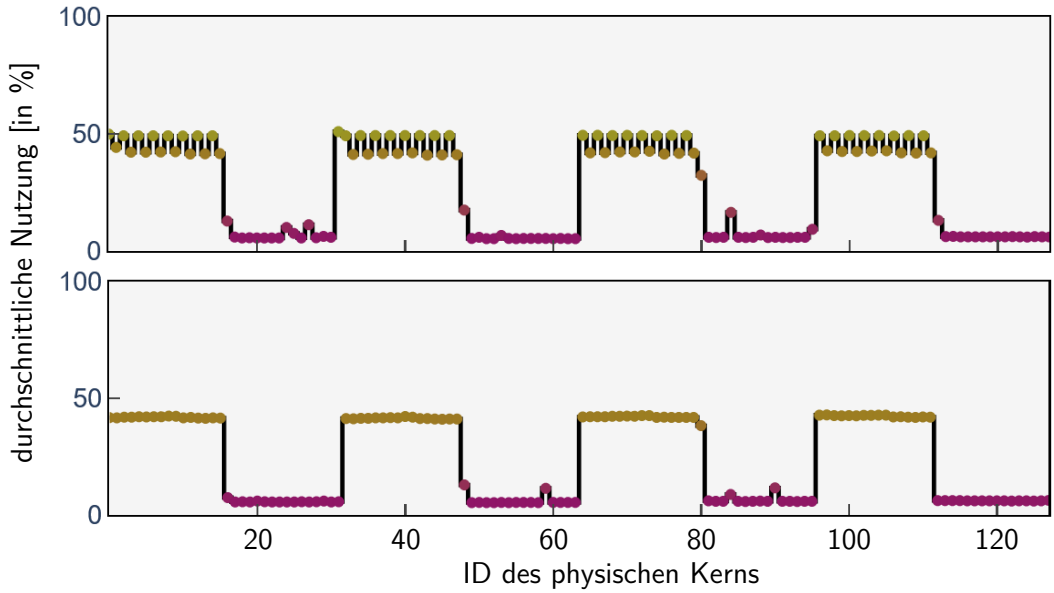

Abbildung 5.3: Nutzung des „ersten“ (oben) und des „zweiten“ (unten) logischen Kerns eines physischen Kerns beim Szenario „Separate NUMAs“ durch HPL und den NVIDIA-HPCG-Benchmark

Die geringsten Auswirkungen auf die Laufzeit von HPL hat Node-Sharing in den Szenarien „Shared CPUs“ und „Shared GPUs“. Dies deutet darauf hin, dass Node-Sharing die Laufzeit am wenigsten beeinflusst, wenn möglichst viele Ressourcen getrennt genutzt werden können. Das bestätigt sich auch dadurch, dass das Szenario „Shared Cores“, in dem die meisten Ressourcen nicht getrennt verwendet werden können, die stärksten Schwankungen in der Laufzeit zeigt. Dieses Szenario ist allerdings auch das Szenario mit der schnellsten Laufzeit ohne Node-Sharing. Daher lässt sich vermuten, dass besonders effiziente Konfigurationen durch die Einführung von Node-Sharing empfindlich beeinträchtigt werden können, was bei einer möglichen Nutzung von Node-Sharing berücksichtigt werden sollte.

Analoge Beobachtungen ergeben sich für die gemessene Rechenleistung des HPL-Benchmarks, die in Abbildung 5.4 ebenfalls in einem Säulendiagramm visualisiert wird. Auch hier zeigt sich, dass Node-Sharing im Regelfall zu einer Verschlechterung führt. Dabei ist die Höhe des Rückgangs der Rechenleistung ebenfalls stark vom gewählten Szenario und der Kombination mit anderen Anwendungen abhängig und reicht von wenigen GFLOP/s bis zu 2021,1 GFLOP/s, was einem relativen Leistungsverlust von bis zu 66,33% entspricht.

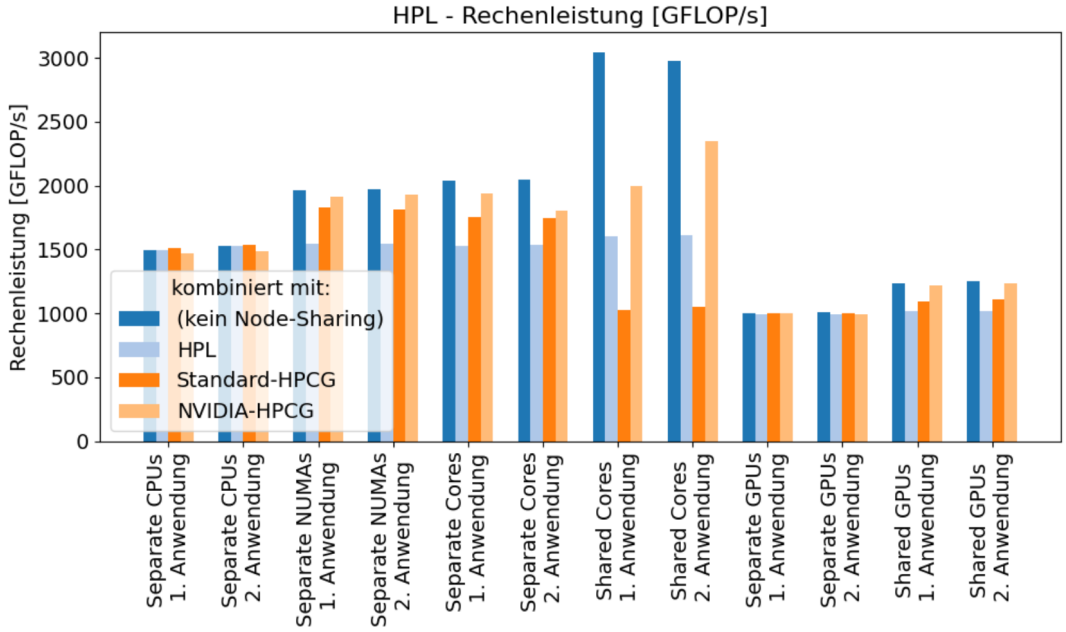

Abbildung 5.4: Rechenleistung des HPL-Benchmarks für verschiedenen Szenarien

Wie bereits bei den Laufzeiten führt auch hier die Kombination von HPL mit sich selbst tendenziell zu den größten Performance-Einbußen, während der geringste Rückgang der Rechenleistung bei der Kombination mit dem NVIDIA-HPCG-Benchmark zu verzeichnen ist. Das bestätigt erneut die aufgestellte Vermutung, dass Node-Sharing die stärksten Auswirkungen zeigt, wenn die Konkurrenz um Ressourcen hoch ist. Allerdings gibt es auch hier,

wie bei der Laufzeit, den Sonderfall im Szenario „Shared Cores“, bei dem die Kombination mit dem Standard-HPCG-Benchmark zu einer deutlich stärkeren Verschlechterung führt als die Kombination mit HPL.

Ebenso bestätigen die Beobachtungen der Rechenleistung, dass die Szenarien „Separate CPUs“ und „Separate GPUs“ scheinbar robust gegenüber Node-Sharing sind. Im klaren Gegensatz dazu reagiert auch hier das Szenario „Shared Cores“ ausgesprochen empfindlich auf Node-Sharing, was sich durch die größten Schwankungen in der Rechenleistung zeigt. Dies ist ebenfalls das Szenario, das ohne Node-Sharing die höchste Rechenleistung erzielt. Demnach kann man auch hier beobachten, dass besonders Konfigurationen, die ohne Node-Sharing die beste Leistung zeigen, stark auf Node-Sharing reagieren.

5.1.2 Auswirkungen von Node-Sharing auf den Standard-HPCG-Benchmark

Danach wurden die Metriken des Standard-HPCG-Benchmarks betrachtet, um zuvor getroffene Vermutungen zu bestätigen und möglicherweise weitere Aussagen treffen zu können.

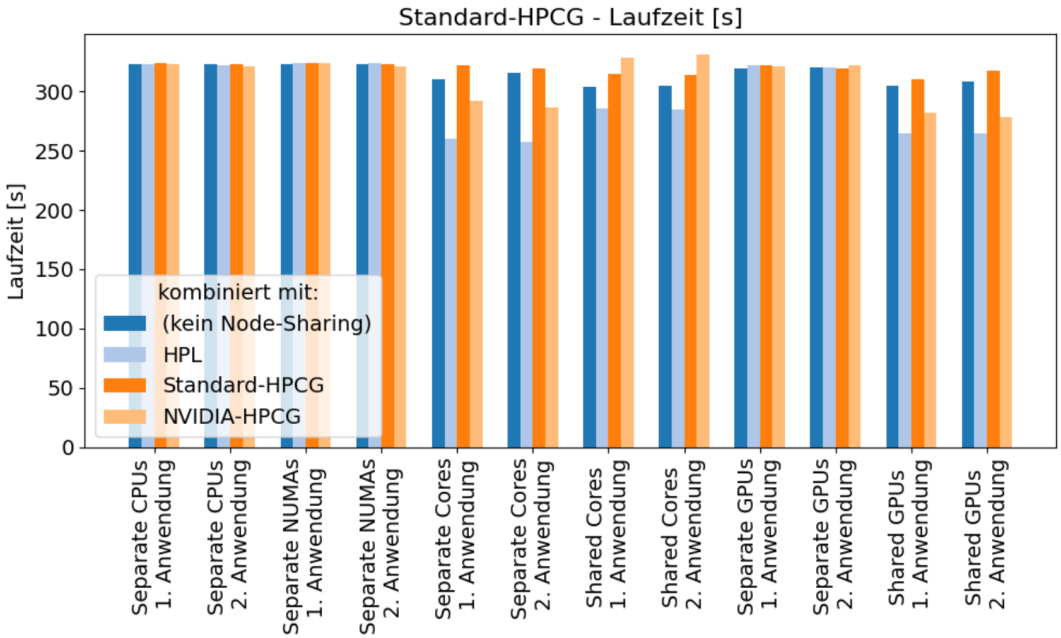

Abbildung 5.5: Laufzeiten des Standard-HPCG-Benchmarks für verschiedene Szenarien

Die Laufzeiten dieses Benchmarks verändern sich in den meisten Szenarien kaum, was darauf zurückzuführen ist, dass in der Eingabedatei des HPCG-Benchmarks eine Laufzeit festgelegt wird. Ein interessantes Verhalten zeigt sich jedoch in einigen Fällen bei der Verwendung von HPL oder dem NVIDIA-HPCG-Benchmark als weitere Anwendung, da sich die Laufzeit um bis zu 58,19 Sekunden (18,45%) verkürzt. Dieser lässt sich allerdings ebenfalls über die in

der Eingabedatei angegebene Laufzeit erklären, da der HPCG-Benchmark nicht überprüft, ob die Laufzeit erreicht wurde. Stattdessen misst er die Zeit, die er für den ersten Satz von CG-Iterationen benötigt, und berechnet anhand dieser, wie viele Sätze von CG-Iterationen er durchführen muss, um die angegebene Laufzeit zu erreichen. Wenn der HPCG-Benchmark im ersten Satz von CG-Iterationen also eine geringere Rechenleistung hat als im weiteren Verlauf, kann dies dazu führen, dass die Gesamtlaufzeit kürzer ausfällt als ursprünglich erwartet. Insgesamt lassen sich daher auf Basis der Laufzeiten bislang nur eingeschränkte Aussagen zu Node-Sharing treffen. [46, 47]

Was allerdings auch hier zu beobachten ist, ist ein robustes Verhalten gegenüber Node-Sharing in den Szenarien „Separate CPUs“ und „Separate GPUs“, da in diesen Szenarien kein starker Rückgang der Laufzeiten zu Beobachten ist. Dies trifft zusätzlich auch auf das Szenario „Separate NUMAs“ zu.

Im Gegensatz zur Laufzeitanalyse zeigt die Betrachtung der Rechenleistung des Standard-HPCG-Benchmarks ein Bild, das den Beobachtungen beim HPL-Benchmark entspricht: Sobald Node-Sharing zum Einsatz kommt, fällt die gemessene Rechenleistung in den meisten Fällen ab (siehe Abbildung 5.6).

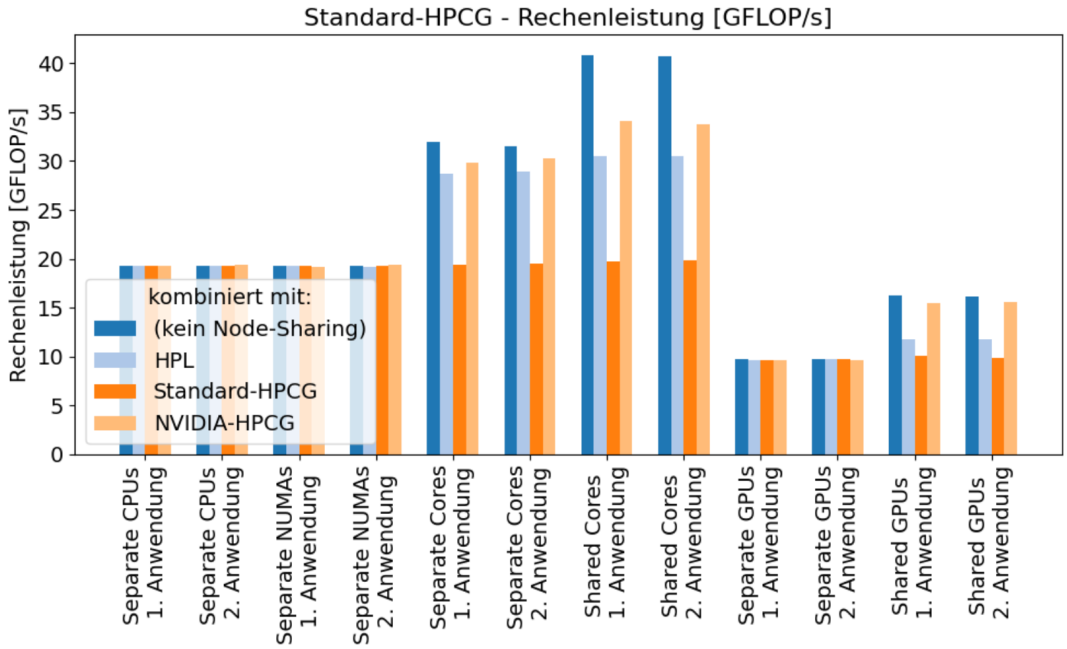

Abbildung 5.6: Rechenleistung des Standard-HPCG-Benchmarks für verschiedene Szenarien

Zu Beobachten ist des Weiteren, dass Kombinationen mit dem Standard-HPCG-Benchmark selbst zu den stärksten Einbrüchen mit bis zu 21,22 GFLOP/s beziehungsweise 51,63% führen, was wie bei HPL vermutlich auf die starke Konkurrenz um Ressourcen bei zwei identischen Anwendungen zurückgeführt werden kann. Demgegenüber führt die Kombination mit dem NVIDIA-HPCG-Benchmark nur zu geringen Einbußen, was ebenfalls wie bei HPL

mit der GPU-lastigen Ressourcenutzung des NVIDIA-Benchmarks und der damit minimalen Nutzung der CPU-Ressourcen zu erklären ist, die vom Standard-HPCG-Benchmark verwendet werden.

Zusätzlich erweisen sich, wie bei den Laufzeiten, die Szenarien „Separate CPUs“, „Separate NUMAs“ und „Separate GPUs“ als sehr robust gegenüber Node-Sharing, da hier die Rechenleistung nahezu unverändert bleibt. Das Szenario „Shared Cores“ hingegen liefert ohne Node-Sharing wie bei HPL die höchste Rechenleistung, welche jedoch mit Node-Sharing stark zurückgeht.

5.1.3 Auswirkungen von Node-Sharing auf den NVIDIA-HPCG-Benchmark

Zuletzt wurden die Auswirkungen von Node-Sharing auf den NVIDIA-HPCG-Benchmark betrachtet, der als einziger der drei Anwendungen hauptsächlich die GPUs verwendet. Die CPU-Kerne hingegen werden von diesem Benchmark kaum genutzt.

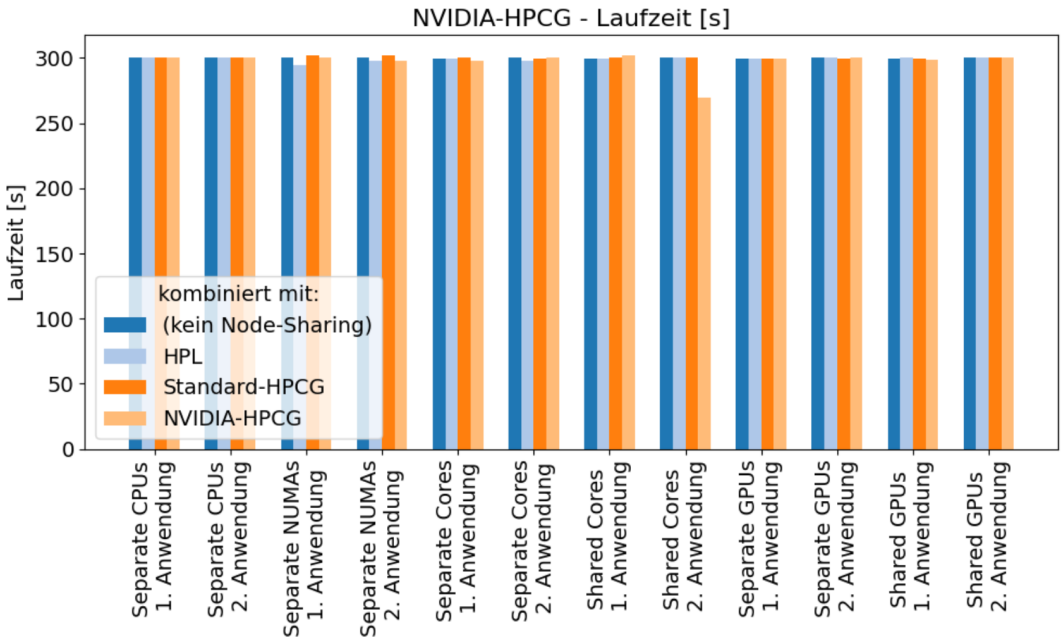

Abbildung 5.7: Laufzeiten des NVIDIA-HPCG-Benchmarks für verschiedene Szenarien

Wie bereits beim Standard-HPCG-Benchmark verändern sich die Laufzeiten des NVIDIA-HPCG-Benchmarks nur sehr wenig. Im Vergleich zu den Laufzeiten ohne Node-Sharing steigt die Laufzeit mit Node-Sharing maximal um 2,65 Sekunden (+0,88%) an und fällt um maximal 5,58 Sekunden (-1,86%) ab. Dieses Verhalten lässt sich, wie beim Standard-HPCG-Benchmark, auf die in der Eingabedatei vorgegebene Laufzeit sowie auf die Art und Weise zurückführen, wie der HPCG-Benchmark diese Zeitvorgabe intern verwendet.

Eine signifikante Abweichung ist in diesem Lauf lediglich, wie in Abbildung 5.7 zu sehen ist, im Szenario „Shared Cores“ zu beobachten, wenn der NVIDIA-HPCG-Benchmark mit sich selbst kombiniert wird. In diesem Fall verkürzt sich die Laufzeit durch Node-Sharing sogar um 30,67 Sekunden, was einem Rückgang von 10,21% entspricht. Auch hier ist der Laufzeit-Rückgang durch die interne Handhabung der vorgegebenen Laufzeit im HPCG-Benchmark zu erklären, die bereits in Kapitel 5.1.2 angesprochen wurde.

Bei der gemessenen Rechenleistung des NVIDIA-HPCG-Benchmarks sind ebenfalls, wie in Abbildung 5.8 zu sehen ist, nur minimale Veränderungen durch Node-Sharing zu erkennen. Sie sind bei weitem nicht so ausgeprägt wie bei HPL oder dem Standard-HPCG-Benchmark. Der maximale Leistungsabfall beträgt gerade einmal 21,09 GFLOP/s, was angesichts einer durchschnittlichen Rechenleistung von etwa 465,703 GFLOP/s relativ gering ist. Dies deutet darauf hin, dass Node-Sharing kaum Auswirkungen auf die Rechenleistung von GPU-lastigen Anwendungen hat, die getrennte GPUs nutzen. Da dies bisher nur für den NVIDIA-HPCG-Benchmark beobachtet wurde, sollte diese Vermutung noch über weitere GPU-lastige Anwendungen bestätigt werden.

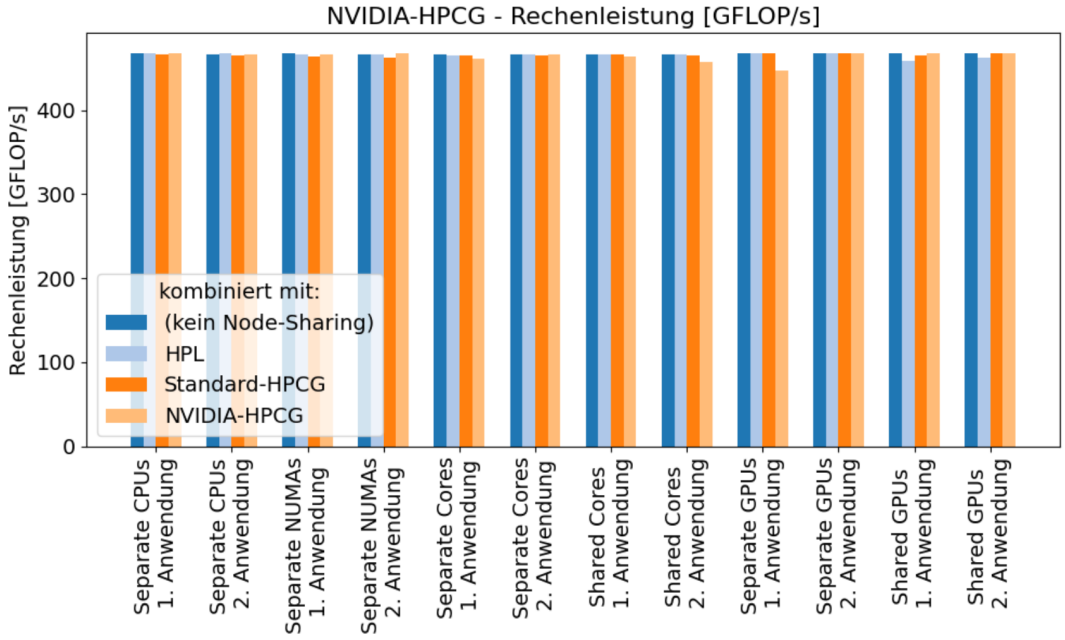

Abbildung 5.8: Rechenleistung des NVIDIA-HPCG-Benchmarks für verschiedene Szenarien

5.2 Analyse durch regelmäßige Ausführung der Testumgebung

Nach der ersten Analyse der Auswirkungen von Node-Sharing wurden insgesamt vier verschiedene Kombinationen aus Anwendungen und Node-Sharing-Szenarien ausgewählt, deren Metriken exemplarisch über mehrere Läufe hinweg betrachtet werden sollten.

Beispielsweise dient die Kombination von HPL mit dem Standard-HPCG-Benchmark im Szenario „Separate CPUs“ als Beispiel für eine Konfiguration, in der Node-Sharing kaum Auswirkungen zeigt und sich die Metriken trotz gleichzeitiger Nutzung eines Knotens im Vergleich zu den Metriken ohne Node-Sharing kaum verändern. Demgegenüber wurde HPL gemeinsam mit sich selbst im Szenario „Separate Cores“ untersucht, um gerade in dieser Konstellation mit zwei identischen Anwendungen zu überprüfen, ob dort die Auswirkungen von Node-Sharing, die in der ersten Analyse zu beobachten waren, auch in weiteren Läufen konstant auftreten. Zusätzlich wurde der NVIDIA-HPCG-Benchmark zusammen mit einer zweiten Instanz desselben Benchmarks für das Szenario „Shared Cores“ ausgeführt, da hier bei der ersten Analyse ein auffälliger Einbruch der Laufzeit auftrat, der in keinem anderen Szenario für den NVIDIA-HPCG-Benchmark zu beobachten war. Obwohl dies vermutlich eine Folge der internen Implementierung von HPCG ist, die in Kapitel 5.1.2 beschrieben wurde, ist es dennoch interessant, ob dieser Einbruch konstant zu beobachten ist oder sich von Lauf zu Lauf unterscheidet. Zuletzt wurde HPL mit dem Standard-HPCG-Benchmark im Szenario „Separate Cores“ betrachtet, um auch für den Standard-HPCG-Benchmark die Frage zu klären, ob die in der ersten Analyse beobachteten Laufzeitrückgänge konstant bleiben oder sich im Zeitverlauf verändern.

Um diese vier Kombinationen weiter untersuchen zu können, wurde die Testumgebung bzw. die implementierte CI/CD-Pipeline in regelmäßigen Abständen ausgeführt. Untersucht wurde die zeitliche Veränderung der Metriken dann mittels LLview, da die Ergebnisse der einzelnen Pipelineläufe im Webportal von LLview gesammelt werden und über die dort gezeigten Graphen einfach zu untersuchen sind.

5.2.1 Analyse von „Separate CPUs“ mit HPL und Standard-HPCG

Die erste betrachtete Konfiguration, HPL mit Standard-HPCG im Szenario „Separate CPUs“, zeigt über alle Läufe hinweg eine bemerkenswerte Stabilität in den gemessenen Metriken. Für beide Programme bleiben die Laufzeiten wie auch die Rechenleistung konstant, was unter anderem in Abbildung 5.9 zu erkennen ist, aber sich auch durch niedrige empirische Variationskoeffizienten widerspiegelt: Für HPL liegt dieser für die Rechenleistung bei lediglich 0.56% und bei der Laufzeit ebenfalls bei nur 0.56%. Auch HPCG zeigt ähnliche Stabilität mit einem Variationskoeffizienten von 0.46% bei der Rechenleistung und 0.47% bei der Laufzeit. Diese geringen Abweichungen zeigen, dass die Metriken für diese Konfiguration sehr stabil bleiben, sodass bereits ein einzelner Lauf, wie bei der ersten Analyse, als repräsentativ für die übrigen gewertet werden kann.

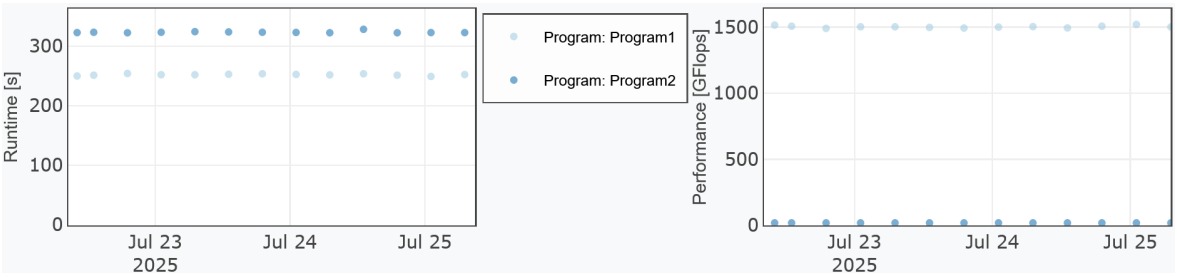

Abbildung 5.9: Metriken für „Separate CPUs“ mit HPL (Program1) und dem Standard-HPCG-Benchmark (Program2)

5.2.2 Analyse von „Separate Cores“ mit zwei Instanzen von HPL

Ein vergleichbar konstantes Verhalten zeigt sich, wenn HPL im Szenario „Separate Cores“ mit sich selbst ausgeführt wird. In Abbildung 5.10 verändern sich Laufzeit und Rechenleistung über alle Läufe hinweg visuell zwar etwas stärker als in der zuvor betrachteten Konfiguration zu beobachten war, allerdings fallen die berechneten empirischen Variationskoeffizienten mit maximal $1,3\%$ insgesamt immer noch sehr niedrig aus. Auch in diesem Fall lässt sich daher die Reproduzierbarkeit der Messergebnisse bestätigen. Die in der ersten Analyse gewonnenen Erkenntnisse basieren demnach nicht auf Ausreißern oder Zufall, sondern spiegeln ein relativ stabil reproduzierbares Verhalten wider.

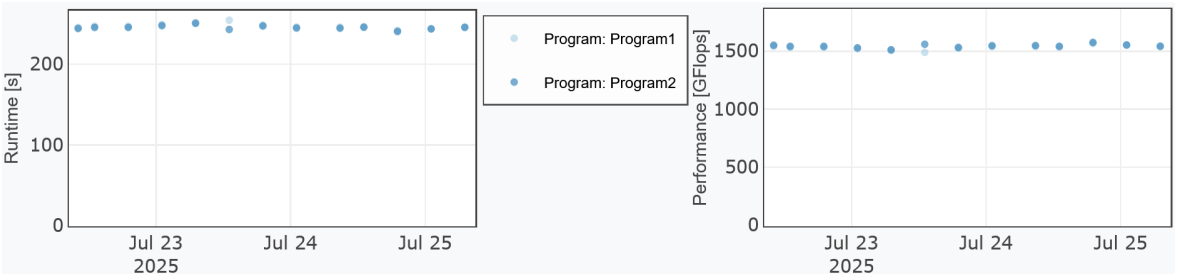

Abbildung 5.10: Metriken für „Separate Cores“ mit zwei Instanzen von HPL (Program1, Program2)

5.2.3 Analyse von „Separate Cores“ mit HPL und Standard-HPCG

Bei der Ausführung von HPL mit dem Standard-HPCG-Benchmark im Szenario „Separate Cores“ zeigen sich ebenfalls konstante Messergebnisse, die in Abbildung 5.11 abgebildet sind. Die berechneten Variationskoeffizienten liegen für alle Metriken unter $1,8\%$ und weisen damit auf eine hohe Reproduzierbarkeit hin. Insbesondere der Standard-HPCG-Benchmark liefert mit einem empirischen Variationskoeffizienten von unter $0,6\%$ sowohl für die Rechenleistung als auch für die Laufzeit sehr stabile Werte. Damit lassen sich die in der ersten Analyse beobachteten Laufzeitrückgänge auch hier bestätigen und als reproduzierbar einstufen.

Interessanterweise verändern sich die Laufzeiten von HPCG trotz der internen Berechnung der benötigten Sätze von CG-Iterationen nicht. Dies legt nahe, dass HPL den HPCG-Benchmark in jedem einzelnen Lauf ähnlich beeinflusst, sodass die einzelnen Iterationssätze in jedem Lauf nahezu gleich lange dauern.

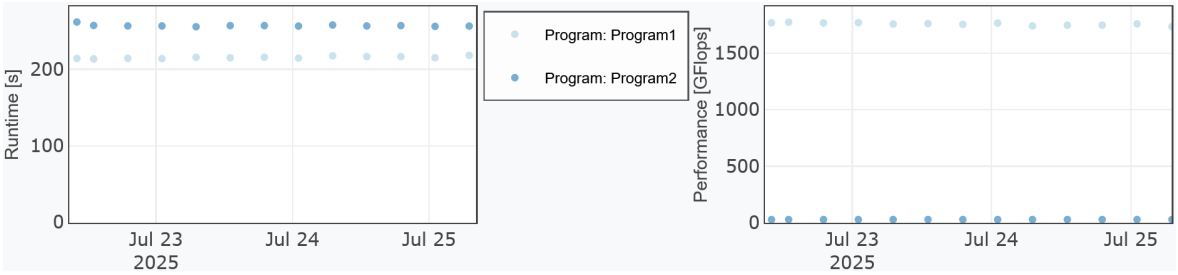

Abbildung 5.11: Metriken für „Separate Cores“ mit HPL (Program1) und dem Standard-HPCG-Benchmark (Program2)

5.2.4 Analyse von „Shared Cores“ mit zwei Instanzen von NVIDIA-HPCG

In der vierten und letzten Konfiguration, zwei NVIDIA-HPCG-Instanzen im Szenario „Shared Cores“, zeigt sich ein deutlich anderes Verhalten. Insbesondere die Laufzeiten variieren hier stark. Während die Variationskoeffizienten für die Rechenleistung mit Werten von 2.26% und 1.03% noch in einem vertretbaren Bereich liegen, zeigt sich bei den Laufzeiten ein klar abweichendes Bild. Sie weisen einen empirischen Variationskoeffizienten von bis zu 8.44% auf, was auf erhebliche Schwankungen zwischen den Läufen hindeutet, die auch in Abbildung 5.12 deutlich sichtbar sind. Diese Schwankungen sind auch im Minimum und Maximum der Laufzeiten zu erkennen: Während einige Läufe nur rund 247 Sekunden benötigen, erreichen andere bis zu 307 Sekunden. Somit ist diese Konfiguration die einzige der vier betrachteten Konfigurationen, bei der keine zuverlässige Reproduzierbarkeit gegeben ist. Da ein einzelner Lauf in diesem Fall nicht repräsentativ für alle anderen Läufe ist, ist es notwendig, die in der ersten Analyse getroffenen Aussagen zu den Auswirkungen von Node-Sharing auf GPU-lastige Anwendungen kritisch zu hinterfragen und weiter zu untersuchen.

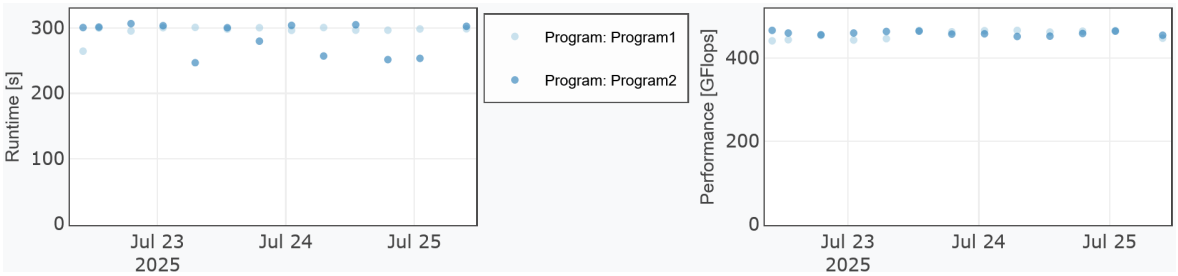

Abbildung 5.12: Metriken für „Shared Cores“ mit zwei Instanzen des NVIDIA-HPCG-Benchmarks (Program1, Program2)

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine automatisierte Testumgebung zur Analyse von Node-Sharing-Szenarien auf Höchstleistungsrechnern entwickelt. Ziel war es, ein flexibles und erweiterbares Werkzeug zu schaffen, das es erlaubt, unterschiedliche Node-Sharing-Szenarien systematisch zu simulieren, auszuführen und auszuwerten. Durch die Kombination von JUBE, einer CI/CD-Pipeline und LLview konnte ein vollständig automatisierter Workflow realisiert werden. Dieser reicht von der Simulation der Szenarien und ihrer Ausführung auf dem HPC-System JURECA über die Erfassung von Metriken, wie Laufzeit und Rechenleistung, bis hin zur grafischen Darstellung der Ergebnisse.

Insgesamt ist die Testumgebung so aufgebaut worden, dass Erweiterungen mit möglichst geringem Aufwand möglich sind: Der größte Teil der Integration neuer Anwendungen oder zusätzlicher Node-Sharing-Szenarien erfolgt beispielsweise ausschließlich über Anpassungen der JUBE-Skripte, während an den anderen Komponenten der Testumgebung kaum Änderungen notwendig sind. Konkret bedeutet dies, dass zur Integration eines neuen Szenarios im JUBE-Skript der Testumgebung ein neues JUBE-Tag ergänzt und das zugehörige Parameterset (vgl. Quelltext 4.4) um das entsprechende Szenario erweitert werden muss. In den JUBE-Skripten der einzelnen Anwendungen muss anschließend noch das anwendungsspezifische Prozess-Pinning für das neue Szenario ergänzt werden. Für neue Anwendungen ist es ähnlich: Sobald ein passendes JUBE-Skript für die Anwendung vorliegt, indem entweder ein bereits existierendes JUBE-Skript wie in Kapitel 4.2.2 angepasst oder ein neues JUBE-Skript für die Anwendung implementiert worden ist, muss das JUBE-Skript der Testumgebung erweitert werden. Dort müssen die entsprechenden JUBE-Tags für die neue Anwendung eingefügt, das Parameterset in Quelltext 4.4 erweitert und die benötigten `<do>`-Elemente in den Steps (vgl. Quelltext 4.3, 4.5) ergänzt werden.

Im Rahmen der Arbeit wurden für eine erste Untersuchung zudem sechs exemplarische Node-Sharing-Szenarien definiert, die sich in der Ressourcennutzung unterscheiden. Erste Analysen mit Benchmark-Anwendungen wie HPL, dem HPCG-Benchmark und der NVIDIA-Variante von HPCG zeigten bereits erkennbare Auswirkungen von Node-Sharing. So führten insbesondere Szenarien mit geteilten physischen Kernen zu erhöhten Laufzeiten und verringerter Rechenleistung. Gleichzeitig deuten andere Szenarien darauf hin, dass durch eine geeignete Trennung der Ressourcen, wie zum Beispiel durch die getrennte Nutzung der CPUs, potenzielle Konflikte zwischen Jobs reduziert werden können. Die Ergebnisse liefern damit

wertvolle Hinweise auf mögliche Konflikte bei der gleichzeitigen Nutzung eines Knotens durch zwei Jobs und unterstreichen dadurch auch die Notwendigkeit, die Auswirkungen von Node-Sharing weiter gezielt zu untersuchen.

Die entwickelte Testumgebung stellt hierfür eine geeignete Grundlage bereit, ist jedoch in ihrem aktuellen Zustand noch mit einigen Einschränkungen verbunden. So ist sie momentan auf ein HPC-System, genauer gesagt JURECA, beschränkt, unterstützt nur die Ausführung von zwei Jobs pro Szenario und nutzt dabei stets nur einen einzelnen Knoten.

Diese Einschränkungen bieten gleichzeitig zahlreiche Ansatzpunkte für zukünftige Erweiterungen. Eine naheliegende Erweiterung besteht in der Unterstützung komplexerer Szenarien. So könnten zukünftig mehr als zwei Anwendungen gleichzeitig berücksichtigt oder mehrere Knoten in einem Szenario verwendet werden. Ein denkbares Beispiel wäre der Einsatz einer GPU-lastigen Anwendung auf den GPUs mehrerer Knoten, während die ungenutzten CPU-Kerne auf mehrere kleinere CPU-lastige Anwendungen verteilt werden. Darüber hinaus könnte die Nutzung der Testumgebung auf weiteren HPC-Systemen eine sinnvolle Erweiterung sein, da unterschiedliche Hardwarearchitekturen einen signifikanten Einfluss auf die Auswirkungen von Node-Sharing haben könnten.

Ebenso ist es sinnvoll, zukünftig weitere Anwendungen und Benchmarks in die Testumgebung zu integrieren, die andere Ressourcen stärker beanspruchen. Dazu zählen beispielsweise I/O-intensive oder kommunikationslastige Anwendungen. Dadurch wäre es möglich, auch für diese Art von Anwendungen verlässliche Aussagen über die Auswirkungen von Node-Sharing zu treffen. Zusätzlich könnten für die bisher verwendeten Anwendungen die genutzten Problemgrößen optimiert werden, um die Ressourcennutzung weiter zu verstärken.

Des Weiteren ist es sinnvoll Performance-Analyse-Tools wie Scalasca [48] oder Score-P [49] in die Testumgebung zu integrieren. Diese Werkzeuge ermöglichen deutlich tiefere Einblicke in das Laufzeitverhalten und die Ressourcennutzung von Anwendungen und könnten dabei unterstützen, Performance-Konflikte durch Node-Sharing noch präziser zu identifizieren. Die Nutzung solcher Tools setzt jedoch eine eingehende Auseinandersetzung mit deren Funktionsweise voraus und erzeugt typischerweise eine große Menge an Performance-Daten, deren Auswertung eine deutlich detailliertere Analyse erfordert, als es im Rahmen dieser Bachelorarbeit möglich war.

Abschließend lässt sich festhalten, dass diese Arbeit mit der entwickelten Testumgebung eine wichtige Basis für die systematische Untersuchung von Node-Sharing geschaffen hat. Über die exemplarische Analyse wurden zudem erste Hinweise auf mögliche Konflikte zwischen

Anwendungen geliefert, die allerdings in ausführlicheren und vertiefenden Untersuchungen genauer analysiert werden müssen. Die entwickelte Testumgebung bietet dazu eine flexibel erweiterbare Grundlage.

Literatur

[1] S. Maloney u. a. „Analyzing HPC Monitoring Data With a View Towards Efficient Resource Utilization“. In: *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 13. Nov. 2024, S. 170–181. DOI: 10.1109/SBAC-PAD63648.2024.00023.

[2] *Hochleistungsrechnen | Speichern und Rechnen | Themen | Forschungsdaten und Forschungsdatenmanagement*. URL: https://forschungsdaten.info/themen/speichern-und-rechnen/hochleistungsrechnen/ (besucht am 26. 05. 2025).

[3] B. Hendrickson und B. Cannon. *ASCR@40: Four Decades of Department of Energy Leadership in Advanced Scientific Computing Research*. Krell Inst., Ames, IA (United States), 25. Sep. 2020. DOI: 10.2172/1665761. URL: https://www.osti.gov/biblio/1665761 (besucht am 26. 07. 2025).

[4] *Profile*. URL: https://www.fz-juelich.de/en/ias/jsc/about-us/profile (besucht am 26. 05. 2025).

[5] *NVIDIA: What is HPC?* NVIDIA. URL: https://www.nvidia.com/en-us/glossary/high-performance-computing/ (besucht am 26. 05. 2025).

[6] *NUMA - HPC Wiki*. URL: https://hpc-wiki.info/hpc/NUMA (besucht am 26. 05. 2025).

[7] *JURECA*. URL: https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/jureca (besucht am 03. 07. 2025).

[8] *Configuration — JURECA user documentation documentation*. URL: https://apps.fz-juelich.de/jsc/hps/jureca/configuration.html (besucht am 03. 07. 2025).

[9] *AMD Rome Processors - HECC Knowledge Base*. URL: https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html (besucht am 25. 06. 2025).

[10] *Processor Affinity — JURECA user documentation documentation*. URL: https://apps.fz-juelich.de/jsc/hps/jureca/affinity.html (besucht am 03. 07. 2025).

[11] *Batch system — JURECA user documentation documentation*. URL: https://apps.fz-juelich.de/jsc/hps/jureca/batchsystem.html (besucht am 03. 07. 2025).

[12] *Slurm Workload Manager - Overview*. URL: https://slurm.schedmd.com/overview.html (besucht am 26. 05. 2025).

[13] *Process distribution, affinity and binding — TGCC public documentation*. URL: https://www-hpc.cea.fr/tgcc-public/en/html/toc/fulldoc/Process_distribution_affinity_binding.html#process-and-thread-affinity (besucht am 26. 05. 2025).

[14] *Slurm Workload Manager - srun*. URL: https://slurm.schedmd.com/srun.html (besucht am 26. 05. 2025).

[15] T. Breuer u. a. *JUBE*. Version REL-2.7.1. Mai 2024. URL: https://doi.org/10.5281/zenodo.11394333.

[16] *JUBE Benchmarking Environment*. URL: https://www.fz-juelich.de/en/ias/jsc/services/user-support/software-tools/jube (besucht am 14. 11. 2024).

[17] *JUBE tutorial — JUBE 2.7.1 documentation*. URL: https://apps.fz-juelich.de/jsc/jube/docu/tutorial.html#installation (besucht am 14. 11. 2024).

[18] *Glossary — JUBE 2.7.1 documentation*. URL: https://apps.fz-juelich.de/jsc/jube/docu/glossar.html (besucht am 14. 11. 2024).

[19] *Command line documentation — JUBE 2.7.1 documentation*. URL: https://apps.fz-juelich.de/jsc/jube/docu/commandline.html (besucht am 15. 11. 2024).

[20] *JUBE/bin/jube-autorun at master · FZJ-JSC/JUBE · GitHub*. URL: https://github.com/FZJ-JSC/JUBE/blob/master/bin/jube-autorun (besucht am 15. 11. 2024).

[21] Y. Müller u. a. *LLview*. 10. Juli 2024. DOI: 10.5281/zenodo.12706843. URL: https://llview.fz-juelich.de (besucht am 11. 07. 2025).

[22] F. Guimaraes, T. Breuer und W. Frings. *Mastering HPC Monitoring Data: From Zero to Hero with LLview*. Meeting Name: ISC High Performance 2024. 2024.

[23] *Benchmarks - User Documentation*. URL: https://apps.fz-juelich.de/jsc/llview/docu/benchmarks/ (besucht am 13. 07. 2025).

[24] *Was ist eine CI/CD-Pipeline?* URL: https://about.gitlab.com/de-de/topics/ci-cd/cicd-pipeline/ (besucht am 26. 05. 2025).

[25] *CI/CD YAML syntax reference | GitLab*. URL: https://docs.gitlab.com/ee/ci/yaml/ (besucht am 20. 11. 2024).

[26] *GitLab Runner | GitLab*. URL: https://docs.gitlab.com/runner/ (besucht am 04. 11. 2024).

[27] *Jacamar CI Runners — JURECA user documentation documentation*. URL: https://apps.fz-juelich.de/jsc/hps/jureca/jacamar.html (besucht am 03. 07. 2025).

[28] *HPL Software*. URL: https://www.netlib.org/benchmark/hpl/software.html (besucht am 10. 07. 2025).

[29] J. J. Dongarra, P. Luszczek und A. Petitet. „The LINPACK Benchmark: past, present and future“. In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), S. 803–820. ISSN: 1532-0634. DOI: 10.1002/cpe.728.

[30] *Home - | TOP500*. URL: https://www.top500.org/ (besucht am 10. 07. 2025).

[31] J. Dongarra, M. A. Heroux und P. Luszczek. „High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems“. In: *The International Journal of High Performance Computing Applications* 30.1 (2016), S. 3–10. DOI: 10.1177/1094342015593158.

[32] A. Aveleda u. a. „Performance Comparison of Scientific Applications on Linux and Windows HPC Server Clusters“. In: Mecánica Computacional. Bd. 29. Nov. 2010.

[33] *hpcg-benchmark/hpcg*. URL: https://github.com/hpcg-benchmark/hpcg (besucht am 08. 07. 2025).

[34] M. Almasri, N. Shustrov und J. Linford. *Accelerating the HPCG Benchmark with NVIDIA Math Sparse Libraries*. NVIDIA Technical Blog. 10. Sep. 2024. URL: https://developer.nvidia.com/blog/accelerating-the-hpcg-benchmark-with-nvidia-math-sparse-libraries/ (besucht am 08. 07. 2025).

[35] *HPCG Benchmark*. URL: https://www.hpcg-benchmark.org/ (besucht am 08. 07. 2025).

[36] *NVIDIA/nvidia-hpcg*. URL: https://github.com/NVIDIA/nvidia-hpcg (besucht am 08. 07. 2025).

[37] *Affinity part 1 – Affinity, placement, and order - AMD GPUOpen*. 17. Apr. 2024. URL: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-affinity-affinity_part1/ (besucht am 26. 07. 2025).

[38] *Bash Reference Manual*. URL: https://www.gnu.org/software/bash/manual/bash.html (besucht am 12. 07. 2025).

[39] *Advanced tutorial — JUBE 2.7.1 documentation*. URL: https://apps.fz-juelich.de/jsc/jube/docu/advanced.html (besucht am 27. 05. 2025).

[40] *OpenSuse Runner :: JSC GitLab documentation*. URL: https://gitlab.pages.jsc.fz-juelich.de/sharedrunner/fixed/ (besucht am 13. 07. 2025).

[41] S. Chacon und B. Straub. *Pro Git*. Apress, 2014. ISBN: 978-1-4842-0077-3 (Softcover), 978-1-4842-0076-6 (eBook). DOI: 10.1007/978-1-4842-0076-6.

[42] *Git - git-remote Documentation*. URL: https://git-scm.com/docs/git-remote#set-url (besucht am 13. 07. 2025).

[43] *Predefined CI/CD variables reference | GitLab Docs*. URL: https://docs.gitlab.com/ci/variables/predefined_variables/#variable-availability (besucht am 13. 07. 2025).

[44] *Stage, commit, and push changes | GitLab Docs*. URL: https://docs.gitlab.com/topics/git/commit/#push-options-for-gitlab-cicd (besucht am 13. 07. 2025).

[45] *Project access tokens | GitLab Docs*. URL: https://docs.gitlab.com/user/project/settings/project_access_tokens/ (besucht am 20. 07. 2025).

[46] *Getting Started | hpcg-benchmark/hpcg*. DeepWiki. URL: https://deepwiki.com/hpcg-benchmark/hpcg/2-getting-started (besucht am 21. 07. 2025).

[47] M. A. Heroux, J. Dongarra und P. Luszczek. *HPCG Benchmark Technical Specification*. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), 1. Okt. 2013. DOI: 10.2172/1113870. URL: https://www.osti.gov/biblio/1113870.

[48] Scalasca developer community. *Toolset for scalable performance analysis of large-scale parallel applications (Scalasca)*. 2. Apr. 2025. DOI: 10.5281/zenodo.15125898.

[49] C. Feld u. a. *Scalable performance measurement infrastructure for parallel codes (Score-P)*. 13. Juli 2025. DOI: 10.5281/zenodo.15873865.