Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Experimentelle Untersuchung zur Optimierung von Spack-Paketen

Jan Julius

jan.julius@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 7412103

Erstgutachter: Dr. Jannek Squar
Zweitgutachter: Professor Dr. Michael Kuhn

Abgabe: 19.6.2026

Inhaltsverzeichnis

1 Einleitung

Die Verwaltung und das Bauen von Softwarepaketen stellt insbesondere in komplexen Systemumgebungen eine anspruchsvolle und zeitaufwendige Aufgabe dar. Dies gilt vor allem für wissenschaftliche Anwendungen und High-Performance-Computing-(HPC)-Umgebungen, in denen zahlreiche Abhängigkeiten, unterschiedliche Compiler sowie plattformspezifische Anforderungen berücksichtigt werden müssen. Spack adressiert viele dieser Herausforderungen, indem es eine flexible und reproduzierbare Installation von Software ermöglicht. Dadurch wird sowohl die Verwaltung als auch der Build-Prozess erheblich vereinfacht. Dennoch erfordert die effiziente Nutzung von Spack weiterhin manuelle Anpassungen. Insbesondere müssen Compiler-Flags, Build-Optionen und weitere Parameter gezielt gewählt und optimiert werden, um eine bestmögliche Performance zu erreichen. Angesichts der Vielzahl möglicher Konfigurationen kann es jedoch schnell schwierig werden, den Überblick zu behalten und systematisch optimale Einstellungen zu identifizieren.

Ziel dieser Bachelorarbeit ist es sich daher mit der experimentellen Analyse und Automatisierung des Erstellens, Bauens und Testens von Spack-Paketen unter Verwendung unterschiedlicher Konfigurationsparameter herauszufinden, welche Parameter einen Einfluss auf die Performance haben und welche nicht.

Da Spack eine sehr große Anzahl von Paketen bereitstellt, wurde der Suchraum für diese Arbeit auf acht ausgewählte Pakete beschränkt. Berücksichtigt wurden die drei MPI-Implementierungen OpenMPI, MPICH und MVAPICH, die drei Compiler GCC, Clang und ICC sowie die beiden mathematischen Bibliotheken OpenBLAS und MKL. Die möglichen Konfigurationen ergeben sich aus den von den Paketen bereitgestellten Variants. Darüber hinaus wurden verschiedene Compiler-Flags als weitere Konfigurationsparameter in die Untersuchung einbezogen.

1.1 Aufbau der Arbeit

Background Dieses Kapitel vermittelt die notwendigen Grundlagen für das Verständnis der restlichen Arbeit und erklärt die Programme, die genutzt wurden.

Relatet Work Es wird einen Überblick über verwandte Arbeiten gegeben.

Benchmarks und Pakete Dieses Kapitel stellt die ausgewählten Spackpakete vor und erklärt die eingesetzten Benchmarks

Design und Implementation Hier wird das entwickelte Programm beschrieben.

Auswertung Die Ergebnisse der Experimente werden analysiert und interpretiert. Es wird untersucht, welche Konfigurationsparameter einen signifikanten Einfluss auf die Performance haben.

Schluss Abschließend werden die wesentlichen Erkenntnisse der Arbeit zusammengefasst, Limitationen reflektiert und mögliche Ansätze für weiterführende Forschung aufgezeigt.

2 Background

2.1 Rechencluster

Rechencluster oder kurz Cluster bezeichnet einen Verbund von eigenständigen Rechenknoten, die über ein Netzwerk miteinander verbunden sind und sich nach außen als ein System präsentieren [Ste02]. Diese Architektur ermöglicht die koordinierte Nutzung verteilter Rechenressourcen, wodurch Probleme bearbeitet werden können, die durch einen einzelnen Rechner nicht effizient gelöst werden könnten.

Strukturell besteht ein typischer HPC-Cluster aus mehreren Komponenten [Ste02].

Rechenknoten (oder auch Nodes) haben den Großteil der Rechenkapazität

Login-Knoten dienen als Zugangspunkt für Nutzer und sind in der Regel nicht für rechenintensive Aufgaben geeignet.

Verwaltungsknoten übernehmen administrative Aufgaben wie Monitoring, Nutzerverwaltung und die Koordination des Gesamtsystems.

Netzwerk für die Kommunikation zwischen den Knoten.

Für die Kommunikation zwischen den Knoten hat sich das Message Passing Interface MPI als Standard etabliert [Ste02]. Da die Prozesse auf verschiedenen Knoten keinen gemeinsamen Adressraum teilen, basiert die Kommunikation ausschließlich auf explizitem Nachrichtenaustausch. Hierzu gibt es in Kapitel 4 mehr.

Da ein Cluster meist von mehreren Personen gleichzeitig genutzt wird, erfolgt die Zuteilung von Rechenzeit durch einen sogenannten Workload-Manager. Einer der am weitesten Verbreiteten ist SLURM.

2.2 SLURM

SLURM (Simple Linux Utility for Resource Management) ist ein Ressourcen-/Workloadmanagementsyste
Es wurde für Hochleistungsrechner entwickelt und wurde mit einigen Designzielen entwickelt [YJG03]:

- Einfachheit: der Quellcode ist einfach genug gehalten, dass er verständlich bleibt und Endnutzer selber Veränderungen vornehmen können
- Open Source: ist frei verfügbar

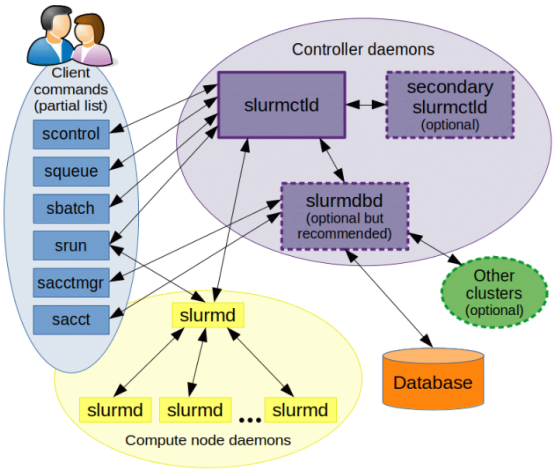

Abbildung 2.1: SLURM Architektur [slu26]

- Portabilität: ist in C geschrieben und lässt sich leicht auf UNIX (ähnliche) Systeme installieren
- Unabhängig vom Netzwerk: unterstützt UDP/IP und andere. Mit Plug-ins können weitere ergänzt werden
- Skalierbarkeit: ist von Grund auf für den Betrieb auf großen Clustern ausgelegt
- Robustheit: Es kann mit verschiedensten Fehlerszenarien umgehen. Ausfälle vom Kontrollknoten oder Rechenknoten: laufende Jobs werden nicht zwingend abgebrochen, und Knoten werden nach Behebung des Fehlers automatisch wieder in den Pool aufgenommen.
- Sicherheit: setzt auf kryptografische Authentifizierung, die über ein Plug-in hinzugefügt werden, um Nutzer und Dienste zu verifizieren.
- Administratorfreundlich: Es nutzt eine einzelne Konfigurationsdatei und Änderungen können jederzeit ohne den laufenden Betrieb zu stören vorgenommen werden.

SLURM hat drei Hauptfunktionen

1. Es weist Ressourcen zu und reserviert diese exklusiv oder nicht exklusiv für einen Nutzer für eine bestimmte Zeit.
2. SLURM stellt ein Framework bereit, mit dem sich Jobs auf Knoten starten, ausführen und überwachen lassen.
3. Jobs kommen in eine Warteschlange, wenn es zu Konflikten bei Ressourcenanfragen kommt.

In Abbildung 2.1 werden die wichtigsten Komponenten von SLURM abgebildet. Auf jedem Rechenknoten läuft ein „slurmd deamon“, die von dem auf dem Verwaltungs-

knoten laufenden „slurmctld deamon“verwaltet werden [slu26]. SLURM Daemons können in sogenannte Partitionen eingeteilt werden. Ein Daemon kann in mehreren Partitionen gleichzeitig vorhanden sein. Eine Aufgabe, die ein Nutzer an SLURM übergibt, wird „Job“genannt und kann in mehrere „Job steps“aufgeteilt sein. Job-Steps können parallel auf einem oder mehreren Knoten ausgeführt werden [slu26].

Im Folgenden werden nur einige für die Arbeit wichtige Befehle besprochen [slu26]. Um das anschaulich zu gestalten, zeige ich dies anhand von Beispielen.

„sinfo“zeigt uns Informationen über die verschiedenen Partitionen und Knoten. Man kann mit weiteren Parametern Sortier-, Formatier- und Filterfunktionen aktivieren.

```
julius@cluster:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
amd          up    6:00:00      3  down* amd[2-4]
magny        up    6:00:00      1  down* magny1
nehalem      up    6:00:00      1  down* nehalem5
west         up    6:00:00      8  alloc west[1-8]
west         up    6:00:00      2  idle  west[9-10]
```

„scontrol“ist ein administratives Werkzeug zum Anzeigen und Ändern des SLURM-Zustands. Viele der Befehle können nur mit Root-Rechten ausgeführt werden. Damit kann man sich z.B. Informationen über eine Node anzeigen lassen.

```
julius@cluster:~$ scontrol show node west9
NodeName=west9 Arch=x86_64 CoresPerSocket=6
   CPUAlloc=0 CPUTot=24 CPULoad=0.00
[...]
   NodeAddr=west9 NodeHostName=west9 Version=19.05.5
   OS=Linux 5.4.0-135-generic #152-Ubuntu SMP Wed Nov 23 20:19:22 UTC 2022
   RealMemory=11000 AllocMem=0 FreeMem=7467 Sockets=2 Boards=1
   State=IDLE ThreadsPerCore=2 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=west
[...]
```

Hier kann man sehen, dass die Node zwei Sockets mit jeweils sechs Cores hat und 2 Threads pro Core und damit sich als 24 CPU insgesamt präsentiert. An RAM hat die Node 11000 MB.

„salloc“weist Ressourcen in Echtzeit zu und startet eine Shell, in der mit srun Jobs parallel ausgeführt werden können. „srun“reicht einen Job ein oder startet Job-Steps. Bietet viele Parameter zur Angabe von Ressourcen oder Eigenschaften von spezifischen Knoten.

„sbatch“reicht ein Job-Skript ein. Wenn genug Ressourcen vorhanden sind, wird es ausgeführt, ansonsten wird es in die Warteschlange gepackt. Meist beinhaltet ein Skript einen oder mehrere „srun“Aufrufe. Im Kopf des Skripts können Informationen für SLURM übergeben werden. Im Beispiel unten sehen wir, dass die Partition und die Anzahl der Nodes, auf denen das Skript laufen soll, übergeben werden. „–job-name“ist dafür da, den Namen des Jobs zu bestimmen.

```
#!/bin/bash
#SBATCH --partition=west
#SBATCH --nodes=2
#SBATCH --job-name=showscript
```

Unser Skript heißt showscript.sh, also übergeben wir mit „sbatch showscript.sh“unser Skript an SLURM. SLURM gibt uns die JobID zurück, unter der unser Programm läuft.

```
    julisu@cluster:~$ sbatch showscript.sh
    Submitted batch job 196117
```

Mit „squeue“können wir uns alle Jobs anzeigen lassen, die gerade laufen oder in der Warteschlange sind.

```
julius@cluster:~$ squeue
        JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        196116      west bench-ru   julius PD       0:00      4 (Resources)
        196117      west showscri   julius  R       0:05      2 west[9-10]
        196109      west bench-ru   julius  R       4:33      4 west[5-8]
        196108      west bench-ru   julius  R       4:36      4 west[1-4]
```

Wie man sehen kann, werden weitere Jobs ausgeführt und einer ist in der Warteschlange. Der Job in der Warteschlange ist vor unserem Job abgegeben worden, aber weil der Job vier Nodes benötigt, konnte er nicht zugeteilt werden. Deshalb hat SLURM unseren Job, der nur zwei Nodes benötigt, vorgezogen, um den Cluster möglichst auszulasten.

Wir brechen unseren Job mit „scancel“ab und schauen noch mal, welche Jobs laufen.

```
julius@cluster:~$ scancel 19117
julius@cluster:~$ squeue
        JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        196116      west bench-ru   julius PD       0:00      4 (Resources)
        196109      west bench-ru   julius  R       5:48      4 west[5-8]
        196108      west bench-ru   julius  R       5:51      4 west[1-4]
```

Die vorgestellten Befehle werden im weiteren Verlauf der Arbeit verwendet, um die verschiedenen Softwarekonfigurationen automatisiert zu bauen und die Benchmarks auszuführen.

2.3 Spack

Spack ist ein Paketmanager, der vor allem im Bereich des Supercomputings genutzt wird. Er dient dazu, Software-Stacks zu erstellen und diese an die jeweilige Systemarchitektur anzupassen. Dabei ermöglicht er den Aufbau konsistenter und reproduzierbarer Umgebungen, insbesondere durch die Nutzung von Environments und Lockfiles [Spa26][GLC+15].

Die Pakete können durch die Angabe von Versionen, Varianten, Compilern und Abhängigkeiten individuell angepasst werden. Dies geschieht weitgehend unabhängig von bereits auf dem System vorhandenen Programmen, wobei bei Bedarf auch externe Pakete

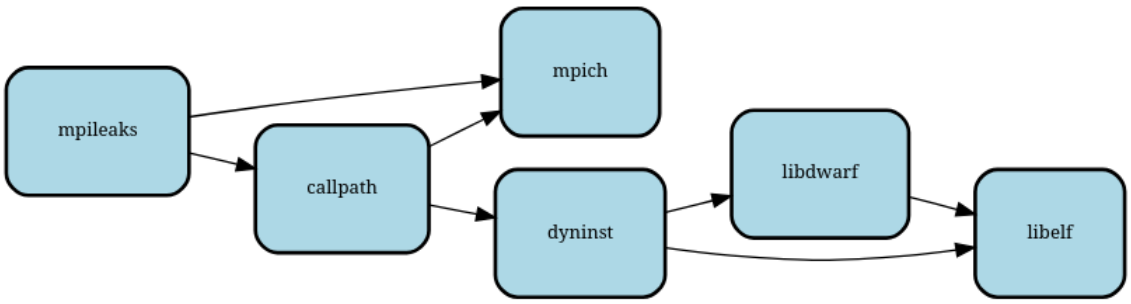

Abbildung 2.2: DAG für mpileaks [Spa26]

eingebunden werden können. Durch die isolierte Installation in separaten Verzeichnissen können mehrere Versionen oder Konfigurationen desselben Programms parallel auf einem System existieren, ohne dass es zu Konflikten kommt.

Spack-Environments dienen dabei der Verwaltung und Spezifikation solcher Software-Stacks. Auf einem Computer oder Cluster können mehrere Environments parallel existieren. Sie sind nicht strikt isolierte Laufzeitumgebungen, sondern wirken vor allem über die aktive Konfiguration (z. B. über spack.yaml) und die vom Nutzer geladene Umgebung.

Wenn Spack ein Paket installiert, erstellt es zunächst einen konkretisierten Abhängigkeitsgraphen (Directed Acyclic Graph, DAG), der alle Varianten, Compiler und Abhängigkeiten berücksichtigt.[GLC+15]

Einige Pakete implementieren gemeinsame Schnittstellen und sind daher austauschbar. In Spack wird dies über sogenannte „virtual packages“realisiert. So implementieren beispielsweise OpenMPI, MVAPICH und MPICH das Paket-Interface „mpi“.

Wenn ein Paket z.B. über „add“angegeben wird, kann man das Paket noch weiter spezifizieren. Spack nutzt dafür eine rekursive Syntax, die in Abbildung 3.3 abgebildet ist [GLC+15][Spa26]:

@ gibt die Version die an die benutzt werden soll

Es können Parameter übergeben werden, die Einfluss darauf haben, wie das Paket gebaut werden soll

% gibt den Compiler an, der genutzt werden soll

^ spezifiziert Abhängigkeiten die genutzt werden sollen

Beim Bauen der Software berücksichtigt Spack diese Einschränkungen und bestimmt selbstständig die freigelassenen Parameter. Dabei kann es vorkommen, dass Spack dieselben Pakete oder Abhängigkeiten mehrfach baut. Wenn sie identisch sind, installiert Spack sie nicht noch einmal, sondern nutzt die bereits vorhandenen.

Spack bietet viele verschiedene Befehle an. Im Folgenden werden einige ausgewählte vorgestellt und die ausgeführten Befehle gekürzt gezeigt[Spa26].

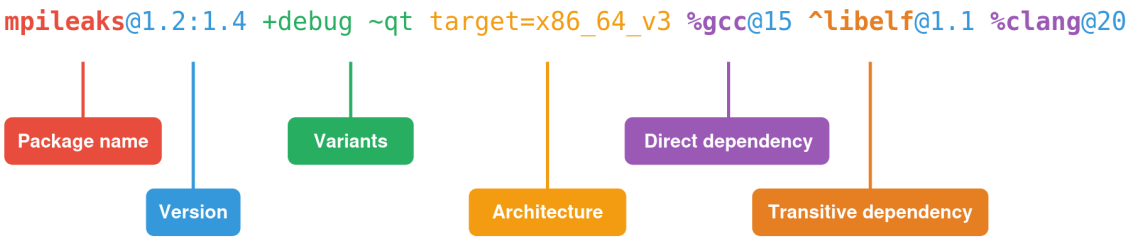

Abbildung 2.3: Spack Syntax [Spa26]

2.3.1 Spack Befehle

Damit Spack überhaupt benutzt werden kann, muss bei jedem Sessionstart Spack geladen werden. In meinem Fall war das:

```
1     source spack/share/spack/setup-env.sh
```

Sobald die Shell den Befehl verarbeitet hat und zurückgekehrt ist, kann man mit Spack arbeiten.

```
1     spack info
```

„spack info {package}“zeigt Informationen über das angegebene Paket an, z. B. verfügbare Versionen, Varianten (Konfigurationsoptionen) sowie Abhängigkeiten.

Listing 2.1: Spack info für MPICH

```
1  user@user:~$ spack info mpich
2  AutoToolsPackage:   mpich
3
4  Description:
5  MPICH is a high performance and widely portable implementation of the
6  Message Passing Interface (MPI) standard.
7
8  Homepage: https://www.mpich.org
9
10 Preferred version:
11     4.3.2 https://www.mpich.org/static/downloads/4.3.2/mpich-4.3.2.tar.gz
12
13 Safe versions:
14     develop    [git] https://github.com/pmodels/mpich.git
15     4.3.2      https://www.mpich.org/static/downloads/4.3.2/mpich-4.3.2.ta
16     r.gz
17
18 [...]
19
20 Deprecated versions:
21     None
22
```

```
Variants:
   argobots [false]                false, true
        Enable Argobots support

    build_system [autotools]        autotools
        Build systems supported by the package

[...]

Licenses:
    mpich2
```

„spack env create {name}“erzeugt ein Environment mit dem gewählten Namen.

```
user@user:~$ spack env create BeispielEnvironment
==> Created environment BeispielEnvironment in: /home/user/spack/var/spack
/environments/BeispielEnvironment
==> Activate with: spack env activate Beispiel
```

„spack env activate {name}“oder „spacktivate {name}“aktiviert ein Environment. Nachfolgende Befehle beziehen sich dann standardmäßig auf dieses Environment.

```
user@user:~$ spacktivate BeispielEnvironment
```

„spack add {package}“fügt dem aktuell aktivierten Environment ein Paket hinzu.

```
user@user:~$ spack add mpich
==> Adding mpich to environment BeispielEnvironment
```

„spack concretize“erzeugt aus der noch abstrakten spack.yaml-Datei eine konkrete Spezifikation aller Pakete. Dabei werden für alle angegebenen Pakete sowie deren Abhängigkeiten Versionen, Varianten, Compiler und weitere Parameter eindeutig festgelegt. Das Ergebnis wird in einer spack.lock-Datei gespeichert. Wenn man sich in einem aktiven Environment befindet, reicht „spack concretize“.

```
user@ueser:~$ spack -e BeispielEnvironment concretize
==> Concretized 1 spec:
[+]  p2m5nk3  mpich@4.3.2~argobots~cuda+fortran~hcoll~hwloc+hydra~level_ze
ro+libxml2+pci~rocm+romio~slurm~vci~verbs+wrapperrpath~xpmem build_system=
autotools datatype-engine=auto device=ch4 netmod=ucx pmi=default platform=
linux os=ubuntu24.04 target=skylake %c,cxx,fortran=oneapi@2025.3.1
[+]  yhvkgqt      ^compiler-wrapper@1.0 build_system=generic platform=linu
x os=ubuntu24.04 target=skylake

[...]

==> Updating view at /home/user/spack/var/spack/environments/BeispielEnvir
onment/.spack-env/view
```

„spack install“installiert alle Pakete, die im aktuell aktivierten oder angegebenen Environment definiert sind. Alternativ kann auch ein einzelnes Paket direkt installiert werden.

```
user@user:~$ spack -e Beispiel install
[+] /usr (external glibc-2.39-thy5vqoaouxdwkh5hofjegtbqsl7r25d)
[+] /home/user/spack/opt/spack/linux-skylake/gcc-runtime-13.3.0-fwwu5g2rr3
krcgzn6dcf4lksko3upl5e
[+] /home/user/spack/opt/spack/linux-skylake/intel-oneapi-runtime-2025.3.1
-uwm24kstubuxgumakljetabzdfh7uhny

[...]

[+] /home/user/spack/opt/spack/linux-skylake/mpich-4.3.2-p2m5nk3qntwgsyl4z
t2juv2vzvm3e3o6
```

3 Related Work

Die Forschungslage zum gewählten Thema ist bislang vergleichsweise dünn. Relevante Vorarbeiten finden sich jedoch in angrenzenden Bereichen, insbesondere zur Compileroptimierung sowie zu verschiedenen MPI-Implementierungen. Sobald Spack in die Betrachtung einbezogen wird, reduziert sich die Anzahl einschlägiger Veröffentlichungen deutlich.

Eine Arbeit, die sich sehr nahe an mein Thema befand, ist „Automated performance analysis of software stacks for distributed systems "[SK23]. Darin werden unterschiedliche Kombinationen aus Compiler, MPI-Implementierung und BLAS-Bibliothek systematisch hinsichtlich ihrer Ausführungseffizienz untersucht und miteinander verglichen. Der dort verfolgte stack-basierte Analyseansatz bildet eine konzeptionelle Grundlage für das eigene Vorgehen.

Im Bereich der Compilerforschung existiert eine breite Grundlagenliteratur. Performanzanalysen liegen sowohl in Form direkter Compiler-Vergleiche als auch als systematische Untersuchungen verschiedener Compiler-Flag-Kombinationen vor [ZH23].

Compiler-Vergleiche auf spezifischer Hardware finden sich etwa in „An Evaluation of the Fujitsu A64FX for HPC Applications "[PDMS+21] , wo mehrere Compiler auf einer ARM-basierten Hochleistungsarchitektur evaluiert werden. Solche Untersuchungen verdeutlichen, dass die Wahl des Compilers je nach Zielhardware erhebliche Auswirkungen auf die erzielte Performance haben kann.

Andere Arbeiten setzen sich nur mit unterschiedlichen MPI Implementationen auseinander.

In „Performance Comparison of Public Domain MPI Implementations using Application Benchmarks "[Yua06] wurden die drei weitverbreiteten Implementierungen MPICH2, OpenMPI und LAM/MPI auf unterschiedlichen Rechenclustern evaluiert, wobei anwendungsnahe Benchmarks als Bewertungsgrundlage dienten.

„Software Architecture and Performance Comparison of MPI/Pro and MPICH "[DS03] setzt Mikro-Benchmarks sowie den LINPACK-Benchmark ein, um Bandbreite und Gleitkommarechenleistung (GFLOPs) der beiden Implementierungen gegenüberzustellen.

Die gesichtete Literatur zeigt insgesamt, dass der Vergleich unterschiedlicher Softwareimplementierungen im HPC-Bereich ein etabliertes Forschungsfeld darstellt. Dennoch fehlt bislang eine systematische Untersuchung, die Compiler, MPI-Implementierungen und das Build-System Spack gemeinsam und unter einheitlichen Bedingungen betrachtet. Eine Lücke, die diese Arbeit adressieren soll.

4 Benchmarks und Pakete

4.1 Benchmarks

4.1.1 OSU-Micro-Benchmarks

OSU-Micro-Benchmarks (OSU) ist eine Sammlung von Tests zur Messung von unterschiedlichen Aspekten von MPI. Es deckt punktuelle Kommunikation aber auch den kollektiven Nachrichtenaustausch ab. Dabei werden unterschiedliche Nachrichtengrößen sowie zahlreiche MPI-Operationen untersucht, um die Kommunikationsleistung eines Systems möglichst umfassend zu analysieren [Bib24].

Besonders Latenz und Bandbreite interessieren uns aus diesen Tests.

4.1.2 HPL

HPL ist eine portable Implementierung des HPLinpack-Benchmarks. Gleichzeitig handelt es sich um ein Softwarepaket, das auf verteilten Computersystemen zufällige dicht besetzte lineare Gleichungssysteme erzeugt, löst, die Korrektheit der Ergebnisse überprüft und die benötigte Rechenzeit misst [DLP03]. HPL erzeugt und löst ein Gleichungssystem der Ordnung n:

$$Ax = b, A \in \mathbb{R}^{n \times n}; x, b \in \mathbb{R}^n \tag{4.1}$$

Dabei ist A eine n×n-Matrix und x sowie b sind Vektoren der Länge n. Zur Lösung des Systems berechnet HPL zunächst eine sogenannte LU-Zerlegung der erweiterten Koeffizientenmatrix [A,b]. Danach kann die Lösung x mit der Gleichung 4.3 berechnet werden [SK23].

$$P_r[A, b] = [[L, U], y], \qquad P_r, L, U \in \mathbb{R}^{n \times n}, \qquad y \in \mathbb{R}^n \tag{4.2}$$

$$Ux = y \tag{4.3}$$

HPL misst die Anzahl der Gleitkommaoperationen pro Sekunde (FLOPS) [Dongarra et al., 2003]. Da pro Rechenoperation vergleichsweise wenige Daten bewegt werden müssen, kann HPL die theoretische Maximalleistung eines Systems nahezu erreichen. Dieses Verhalten ist jedoch nur bedingt repräsentativ für reale Anwendungen, da diese häufig stärker durch Speicherzugriffe als durch reine Rechenleistung begrenzt sind [SK23].

HPL kann über verschiedene Parameter konfiguriert werden. Die wichtigsten davon sind [Dongarra et al., 2003]:

- N: Größe der Koeffizientenmatrix

- NB: Blockgröße, also die Größe der Teilblöcke, in die die Matrix zerlegt wird
- P und Q: Dimensionen des Prozessrasters, das verwendet wird, um die Blöcke zyklisch auf die Prozesse zu verteilen

4.1.3 HPCC

Die HPC Challenge Benchmark Suite (HPCC) ist eine Sammlung von Benchmarks, die eine realistischere Leistungsmessung ermöglichen. Dafür untersucht HPCC:

- Effizienz paralleler Berechnungen
- Speicherbandbreite
- Speicherlatenz
- Kommunikationsgeschwindigkeit zwischen Prozessoren
- Verhalten bei zufälligen Speicherzugriffen

Diese werden von mehreren Tests abgedeckt:

- HPL für die maximale Rechenleistung
- STREAM für Speicherbandbreite
- RandomAccess für zufällige Speicherzugriffe
- PTRANS für Netzwerk- und Kommunikationsleistung
- FFT für Fourier-Transformationen
- DGEMM für Matrixmultiplikationen
- Latency/Bandwidth für Kommunikationslatenz und Bandbreite

Die Benchmarks können nur auf einem, mehreren separaten oder mehreren miteinander kommunizierenden Prozessoren ausgeführt werden. Damit Cluster unabhängig von ihrer Größe getestet werden können, können die Problemgrößen angepasst werden [DTB+04].

4.2 Pakete

4.2.1 MPI

MPI, kurz für Message Passing Interface, ist, wie der Name schon sagt, ein Interface, das die Nachrichtenübertragung zwischen Prozessen standardisiert. Funktionen wie das Senden und Empfangen von Nachrichten und Synchronisation zwischen einzelnen Prozessoren oder Gruppen. Es wird vor allem im HPC-Bereich eingesetzt, damit mehrere

Prozesse an einer Aufgabe mitarbeiten können. Es existieren mehrere Implementationen. [GLS99]

In dieser Arbeit habe ich mich auf OpenMPI, MPICH und MVAPICH konzentriert. Für alle Implementationen wurde eine Auswahl an „Variants“getroffen. Die Listen aller Variants und welche genommen wurden, sind auf den folgenden Seiten Tabelle 4.1.

4.2.2 Compiler

Um ein Programm ausführen zu können, braucht Spack einen Compiler, der ein Programm in Assembly übersetzt und damit grundsätzlich erstmal ausführbar macht. Jedes andere Programm braucht einen Compiler, damit es funktionieren kann. Die Art und Weise, wie ein Compiler den Code übersetzt, ist von Compiler zu Compiler unterschiedlich und damit auch die Performance. Compiler können für eine bestimmte Architektur spezialisiert sein.[GRB+12]

Die Compiler die für dieses Paper verwendet wurden sind: gcc, clang bzw. llvm und icc bzw. intel-oneapi-compilers

Auf den folgenden Seiten sind Tabellen für die gewählten Variants für llvm und gcc. Für intel-oneapi-compilers habe ich nur envmods gewählt, weil es die einzige Variant war, welche nicht im direkten Konflikt mit dem benutzten Cluster stand.

4.2.3 BLAS

Basic Linear Algebra Subprograms oder kurz BLAS ist ein Interface für Mathebibliotheken, die lineare Algebra implementieren. Da mathematische Berechnungen stark mit der Hardwarearchitektur zusammenhängen, sollte auch dies getestet werden. Es wurden OpenBLAS und intel-oneapi-mkl (oder kurz mkl) ausgesucht. Alternativen sind zum Beispiel ATLAS oder BLIS.

Im Folgenden sind Tabellen mit den gewählten Paketen und Spack-Variants.

Variant	Default	Condition	Description	genommen	Begründung
atomics	true	–	Enable built-in atomics	ja	notwendig für performante atomare Operationen
cray-xpmem	false	fabrics=xpmem	Use cray-xpmem instead of xpmem configure flag	nein	keine Cray-Systeme
cuda	false	–	Build with CUDA	nein	kein CUDA/GPU-aware MPI benötigt
cxx	false	@:4	Enable deprecated C++ MPI bindings	nein	deprecated
cxx_exceptions	false	@:4	Enable deprecated C++ exception support	nein	deprecated
debug	false	build_system=autotools	Make debug build	nein	reduziert Performance
fortran	true	–	Enable Fortran support	ja	Fortran-Unterstützung für HPC-Anwendungen sinnvoll
gpfs	false	–	Enable GPFS support	-	-
internal-hwloc	false	–	Use internal hwloc	-	-
internal-libevent	false	–	Use internal libevent	-	-
internal-pmix	false	@3:	Use internal pmix and prrte	-	-
ipv6	false	@4:	Enable IPv6 support	-	-
java	false	@1.7.4:	Build Java support	-	-
legacylaunchers	false	@1.6:4 schedulers=slurm	Do not remove mpirun/mpiexec when building with slurm	nein	in kombination mit slurm
lustre	false	–	Lustre filesystem library support	-	-
memchecker	false	–	Memchecker support for debugging	nein	verursacht Performance-Overhead
openshmem	false	–	Enable building OpenSHMEM	-	-
orterunprefix	false	@1.3:4	Prefix Open MPI to PATH and LD_LIBRARY_PATH	-	-
pmi	false	@1.5.5:4 schedulers=slurm	Enable PMI support	ja	bessere Integration mit SLURM möglich
rocm	false	–	Enable ROCm support	nein	keine AMD-GPUs vorhanden
romio	true / false	@:5 / @5:	Enable ROMIO support	ja	paralleles MPI-I/O sinnvoll
rsh	true	–	Enable rsh (OpenSSH) process lifecycle management	ja	-
sqlite3	false	@1.7.3:1	Build SQLite3 support	-	-
static	false	–	Build static libraries	-	-
thread_multiple	false	@1.5.4:2	Enable MPI_THREAD_MULTIPLE support	-	-
two_level_namespace	false	–	Build shared libraries with two-level namespace	nein	hauptsächlich macOS-spezifisch
vt	true	–	Build VampirTrace support	nein	Profiling erzeugt zusätzlichen Overhead
wrapper-rpath	true	@1.7.4:	Enable rpath support in the wrappers	ja	erleichtert konsistente Laufzeitumgebung

Tabelle 4.1: OpenMPI binäre Variants

Variant	Default	Choices	Condition	Description	genommen	Begründung
amdgpu_target	none	gfx1010, gfx1011, gfx1012, gfx1013, gfx1030, gfx1031, gfx1032, gfx1033, gfx1034, gfx1035, gfx1036, gfx1100, gfx1101, gfx1102, gfx1103, gfx701, gfx801, gfx802, gfx803, gfx900, gfx902, gfx904, gfx906, gfx908, gfx909, gfx90a, gfx90c, gfx940, gfx941, gfx942	+rocm	AMD GPU architecture	nein	Rechner laufen überwiegend CPU-basiert, kein ROCm-Einsatz
build_system	autotools	autotools	–	Build systems supported by the package	ja	Standard-Buildsystem von OpenMPI
cuda_arch	none	10–90a	+cuda	CUDA architecture	nein	keine NVIDIA-GPU-Beschleunigung notwendig
fabrics	none	auto, cma, fca, hcoll, knem, mxm, ofi, psm, psm2, ucc, ucx, verbs, xpmem	–	Enabled fabrics	ja (ucx)	-
romio-filesystem	none	daos, gpfs, ime, lustre, nfs, panfs, pvfs2, quobytefs, testfs, ufs, xfs	–	Add the filesystem to romio	-	-
schedulers	none	alps, auto, loadleveler, lsf, sge, slurm, tm	–	Enabled schedulers	nein	slurm, erzeugte Fehlermeldungen

Tabelle 4.2: Nicht-binäre Variants von OpenMPI

Variant	Default	Choices	Condition	Description	genommen	Begründung
amdgpu_target	none	none, gfx1010, gfx1011, gfx1012, gfx1013, gfx1030, gfx1031, gfx1032, gfx1033, gfx1034, gfx1035, gfx1036, gfx1100, gfx1101, gfx1102, gfx1103, gfx701, gfx801, gfx802, gfx803, gfx900, gfx900:xnack-, gfx902, gfx904, gfx906, gfx906:xnack-, gfx908, gfx908:xnack-, gfx909, gfx90a, gfx90a:xnack+, gfx90a:xnack-, gfx90c, gfx940, gfx941, gfx942	+rocm	AMD GPU architecture		
build_system	autotools	autotools	–	Build systems supported by the package		
cuda_arch	none	none, 10, 100, 100a, 100f, 101, 101a, 101f, 103, 103a, 103f, 11, 110, 110a, 110f, 12, 120, 120a, 120f, 121, 121a, 121f, 13, 20, 21, 30, 32, 35, 37, 50, 52, 53, 60, 61, 62, 70, 72, 75, 80, 86, 87, 89, 90, 90a	+cuda	CUDA architecture		
datatype-engine	auto	auto, dataloop, yaksa	@3.4:	Controls the datatype engine to use		
device	ch4	ch3, ch3:sock, ch4	–	Abstract Device Interface (ADI) implementation		
netmod	ofi	mxm, ofi, tcp, ucx	–	Network module		
pmi	default	cray, default, pmi, pmi2, pmix	–	PMI interface		

Tabelle 4.3: Nicht-binäre Variants von mpich

Variant	Default	Condition	Description	genommen	Begründung
argobots	false	–	Enable Argobots support	-	-
cuda	false	–	Build with CUDA	nein	kein CUDA-aware MPI erforderlich
fortran	true	–	Enable Fortran support	ja	Fortran ist wichtig für viele HPC-Anwendungen
hcoll	false	@3.3: device=ch4 netmod=ucx	Enable support for Mellanox HCOLL accelerated collective operations library	nein	-
hwloc	true	–	Use external hwloc package	ja	wichtig für NUMA- und Topologie-Erkennung????
hydra	true	–	Build the hydra process manager	ja	Standard-Prozessmanager für MPICH ?????
level_zero	false	–	Enable level zero support	-	-
libxml2	true	–	Use libxml2 for XML support instead of the custom minimalistic implementation	ja	stabile externe XML-Implementierung bevorzugt????
pci	true	–	Support analyzing devices on PCI bus	ja	hilfreich für Geräte- und Topologie-erkennung
rocm	false	–	Enable ROCm support	nein	keine AMD-GPUs vorhanden
romio	true	–	Enable ROMIO MPI I/O implementation	ja	paralleles MPI-I/O für HPC sinnvoll
slurm	false	–	Enable Slurm support	ja	Integration in vorhandenen SLURM-Scheduler
vci	false	@4: device=ch4	Enable multiple VCI critical sections for concurrent MPI communications	-	-
verbs	false	–	Build support for OpenFabrics verbs	nein	UCX/OFI moderner und flexibler
wrapperrpath	true	–	Enable wrapper rpath	ja	konsistente Laufzeitumgebung
xpmem	false	@3.4:	Enable XPMEM support	-	-

Tabelle 4.4: binäre Variants von MPICH

Variant	Default	Choices / Condition	Description	genommen	Begründung
alloca	false	Use `alloca` to allocate temporary memory if available	Verwendet `alloca` für temporäre Speicherallokation	-	-
debug	false	Enable debug info and error messages at run-time	Aktiviert zusätzliche Debug-Informationen und Laufzeitprüfungen	nein	Performance wird dadurch vermutlich schlechter
regcache	true	Enable memory registration cache	Aktiviert Cache für Speicherregistrierung bei RDMA-Kommunikation	ja	Verbessert Performance bei Hochgeschwindigkeitsnetzwerken?????
wrapperrpath	true	Enable wrapper rpath	Fügt automatisch RPATH-Einträge in Wrapper-Compiler ein	ja	Vereinfacht Nutzung der MPI-Bibliotheken
build_system	autotools	autotools	Build systems supported by the package	ja	Standard-Buildsystem des Pakets
file_systems	auto	auto, gpfs, lustre, nfs, ufs	List of the ROMIO file systems to activate	auto	Automatische Erkennung ausreichend ????
netmod	ofi	ofi, ucx	Select the netmod to be enabled for this build. For IB/RoCE systems, use the `ucx` netmod; for interconnects supported by libfabric, use the `ofi` netmod	ja	-
process_managers	auto	none, auto, gforker, hydra, remshell, slurm	List of the process managers to activate	slurm	Integration in vorhandene SLURM-Umgebung
threads	multiple	funneled, multiple, serialized, single	Control the level of thread support	multiple	Unterstützung paralleler Multi-Thread-Anwendungen

Tabelle 4.5: MVAPICH Variants

Variant	Default	Choices / Condition	Description	genommen	Begründung
binutils	false	false, true	Build via binutils	ja	Verwendung externer Binutils für bessere Toolchain-Integration
bootstrap	true	false, true	Enable 3-stage bootstrap	ja	Erhöht Zuverlässigkeit und Konsistenz des Compilers
build_system	autotools	autotools	Build systems supported by the package	ja	Standard-Buildsystem von GCC
build_type	RelWithDebInfo	Debug, MinSizeRel, RelWithDebInfo, Release	CMake-like build type. Debug: `-O0 -g`; Release: `-O3`; RelWithDebInfo: `-O2 -g`; MinSizeRel: `-Os`	ja	-
graphite	false	false, true	Enable Graphite loop optimizations (requires ISL)	ja	-
languages	c,c++,fortran	ada, brig, c, c++, d, fortran, go, java, jit, lto, obj-c++, objc	Compilers and runtime libraries to build	c,c++,fortran	Benötigte Sprachen für wissenschaftliche HPC-Anwendungen
mold	false	false, true (`@12:`)	Use mold as the linker by default	nein	Kein Bedarf für alternativen Linker
nvptx	false	false, true	Target nvptx offloading to NVIDIA GPUs	nein	Keine NVIDIA-GPU-Offloading-Unterstützung notwendig
piclibs	false	false, true	Build PIC versions of `libgfortran.a` and `libstdc++.a`	-	-
profiled	false	false, true (`+bootstrap`)	Use Profile Guided Optimization	-	-
strip	false	false, true	Strip executables to reduce installation size	ja	Reduziert Speicherbedarf der Installation

Tabelle 4.6: GCC Variants

Variant	Default	Condition	Description	genommen	Begründung
clang	true	–	Build the LLVM C/C++/Objective-C compiler frontend	ja	Hauptbestandteil der LLVM-Toolchain
code_signing	false	+lldb platform=darwin	Enable code-signing on macOS	nein	Kein macOS
cuda	false	–	Build with CUDA	nein	Keine CUDA-basierte GPU-Beschleunigung vorgesehen
flang	false	–	Build the LLVM Fortran compiler frontend	ja	Fortran-Unterstützung für wissenschaftliche Anwendungen
gold	true	–	Add support for LTO with the gold linker plugin	ja	Unterstützung für Link-Time-Optimization
ipo	false	build_system=cmake	CMake interprocedural optimization	-	-
libomp_tsan	false	[c]*make*@3.9 : @4:12	Build with OpenMP capable thread sanitizer	nein	Kein Fokus auf Thread-Debugging
libomptarget	true	–	Build the OpenMP offloading library	ja	-
libomptarget_debug	false	–	Allow debug output with the environment variable LIBOMPTARGET_DEBUG=1	nein	Debug verringert vermutlich die Performance
link_llvm_dylib	false	+llvm_dylib	Link LLVM tools against the LLVM shared library	-	-
lld	true	–	Build the LLVM linker	ja	Moderner und schneller Linker
lldb	true	–	Build the LLVM debugger	nein	debugger verringert vermutlich die Performance
llvm_dylib	true	–	Build a combined LLVM shared library	ja	Reduziert Speicherbedarf und vereinfacht Linking
lua	true	–	Enable lua scripting inside lldb	ja	-
mlir	false	@10:	Build with MLIR support	-	-
offload	true	@19:	Build the Offload subproject	ja	-
polly	true	–	Build the LLVM polyhedral optimization plugin	ja	Zusätzliche Schleifenoptimierungen
python	false	–	Install python bindings	-	-
split_dwarf	false	–	Build with split dwarf information	nein	Debug erzeugt vermutlich Performanceverlust
utils	false	–	Install utility binaries (FileCheck, etc.)	ja	-
z3	false	–	Use Z3 for the clang static analyzer	-	-
zstd	false	@15:	Enable zstd support for static analyzer / lld	ja	Effizientere Kompression und schnellere Verarbeitung

Tabelle 4.7: binäre Variants von LLVM

Variant	Default	Choices	Condition	Description	genommen	Begründung
build_system	cmake	cmake	–	Build systems supported by the package	ja	Standard-Buildsystem von LLVM
build_type	Release	Debug, MinSizeRel, RelWithDebInfo, Release	build_system=cmake	CMake build type	Release, MinSizeRel	Optimierte Produktions-Builds
compiler-rt	runtime	none, project, runtime	–	Build the LLVM compiler runtime including sanitizers	runtime??????	Benötigte Runtime-Komponenten für Clang
cuda_arch	none	10, 20, 30, 35, 50, 60, 70, 75, 80, 86, 89, 90, 90a	+cuda	CUDA architecture	none	CUDA deaktiviert
generator	ninja	none	build_system=cmake	Build system generator to use	ninja????????	Schnellere und effizientere Builds
libcxx	runtime	none, project, runtime	–	Build the LLVM C++ standard library	runtime??????	Standardkonfiguration für Clang-Runtimes
libunwind	runtime	none, project, runtime	–	Build the LLVM unwinder library	runtime???????	Standard-Runtime-Konfiguration
openmp	runtime	project, runtime	–	Build OpenMP support	runtime????????	OpenMP-Runtime für parallele Anwendungen
shlib_symbol_version	none	none	@13:	Add shared library symbol version	none	-
targets	all	aarch64, all, amdgpu, arm, avr, bpf, hexagon, mips, nvptx, powerpc, riscv, sparc, systemz, webassembly, x86	–	What targets to build	x86????	Zielarchitektur der vorhandenen Systeme
version_suffix	none	none	–	Add a symbol suffix	-	-

Tabelle 4.8: Nicht-binäre Variants von LLVM

Variant	Default	Choices	Condition	Description	genommen	Begründung
bignuma	false	false, true	–	Enable experimental support for up to 1024 CPUs/Cores and 128 numa nodes	-	-
build_system	makefile	cmake, makefile	–	Build systems supported by the package	ja	-
build_type	Release	Debug, MinSizeRel, RelWithDebInfo, Release	build_system=cmake	CMake build type	ja	Release und MinSizeRel könnten interessant sein
consistent_fpcsr	false	false, true	–	Synchronize FP CSR between threads (x86/x86_64 only)	nein	leichter zusätzlicher Overhead ohne relevanten Nutzen
dynamic_dispatch	true	false, true	–	Enable runtime cpu detection for best kernel selection	ja	optimale Kernelwahl abhängig von der CPU-Architektur
fortran	true	false, true	@0.3.21:	Build LAPACK/Fortran support	ja	-????
generator	make	none	build_system=cmake	The build system generator to use	–	–
ilp64	false	false, true	–	Force 64-bit Fortran native integers	-	-
ipo	false	false, true	build_system=cmake ĉmake@3.9:	CMake interprocedural optimization	nein	nicht relevant bei Nutzung des makefile-Buildsystems
locking	true	false, true	–	Build with thread safety	ja	Thread-Sicherheit für parallele Anwendungen notwendig
pic	true	false, true	–	Build position independent code	ja	erforderlich für Shared Libraries und flexible Nutzung
shared	true	false, true	–	Build shared libraries	ja	geringerer Speicherverbrauch und bessere Integration
symbol_suffix	none	none	–	Set a symbol suffix	nein	keine parallele Installation mehrerer OpenBLAS-Versionen erforderlich
threads	none	none, openmp, pthreads	–	Multithreading support	ja (openmp)	OpenMP bietet effiziente parallele BLAS-Ausführung

Tabelle 4.9: Openblas Variants

Variant	Default	Choices	Condition	Description	genommen	Begründung
build_system	generic	generic	–	Build systems supported by the package	ja	Standard-Buildsystem des Pakets
cluster	false	false, true	–	Build with cluster support (scalapack, blacs, etc.)	ja	Cluster-Funktionalität für HPC erforderlich
envmods	true	false, true	–	Toggles environment modifications	ja	Automatische Umgebungsanpassung
gfortran	false	false, true	–	Compatibility with GNU Fortran	ja	Kompatibilität mit GNU-Fortran-Toolchain
ilp64	false	false, true	–	Build with ILP64 support	nein	64-bit Integer nicht benötigt
mpi_family	none	none, mpich, openmpi	–	MPI family	openmpi und mpich	Integration in bestehende OpenMPI-Umgebung
shared	true	false, true	–	Builds shared library	ja	Bessere Speicher- und Link-Effizienz
threads	none	none, openmp, tbb	–	Multithreading support	openmp	OpenMP wird für parallele Anwendungen benötigt

Tabelle 4.10: MKL Variants

5 Design und Implementation

Mit den im vorherigen Kapitel ausgewählten Paketen, Varianten und Benchmarks, muss eine Strategie zur Durchsuchung des Suchraums festgelegt werden. Der naheliegendste Ansatz ist, alle möglichen Environments zu erzeugen und diese vollständig zu testen. Die grundlegende Problematik besteht jedoch darin, dass der Suchraum sehr groß ist: Er umfasst drei MPI-Implementierungen, drei Compiler und zwei Mathematikbibliotheken, von denen jedes Paket eine Vielzahl boolescher und kategorischer Varianten besitzt. Dadurch wächst die Anzahl möglicher Kombinationen exponentiell. Aufgrund dieses großen Suchraums sowie der zeitaufwendigen Evaluierung einzelner Konfigurationen ist eine vollständige Exploration nicht praktikabel. Stattdessen wird eine Teilmenge des Suchraums untersucht und daraus eine möglichst gute Lösung abgeleitet, wobei in Kauf genommen wird, dass die global optimale Lösung unter Umständen nicht gefunden wird. Um dennoch ein möglichst gutes Ergebnis zu erzielen, wird ein Machine-Learning-basierter Ansatz verwendet. Das Problem wird als Optimierungsproblem aufgefasst, bei dem die Performanz eines Stacks die Zielfunktion darstellt. Ziel ist es, eine Konfiguration zu finden, die diese Zielfunktion maximiert. Das Programm ist in fünf Python-Module aufgeteilt:

- optimizer: Steuerprogramm, das die übergeordnete Optimierungsschleife implementiert sowie das Laden und Speichern von Checkpoints übernimmt
- config_builder: übersetzt eine Konfiguration in Spack-Befehle und baut das zugehörige Spack-Environment
- config_proposer: definiert den Suchraum, erzeugt neue Konfigurationen und implementiert die Explore-Exploit-Strategie
- config_evaluator: orchestriert den vollständigen Evaluierungsablauf und definiert die Ergebnisdatenstruktur EvalResult
- benchmarks: kapselt den Bau der Benchmark-Binaries, die Ausführung über SLURM und das Parsen der Ergebnisse

5.1 Konfigurationsraum

Der Suchraum ist in config_proposer.py in der Datenstruktur SEARCH_SPACE definiert. Er umfasst drei Dimensionen:

- **MPI**: mpich, openmpi@4 und mvapich@4.0, jeweils mit ihren spezifischen booleschen Varianten (z.,B. +fortran, ~verbs) und kategorischen Varianten (z.,B. device=ch4, netmod=ofi, schedulers=slurm).
- **Compiler**: gcc, llvm und intel-oneapi-compilers, ebenfalls mit booleschen Varianten (z.,B. +clang, +polly) und kategorischen Varianten (z.,B. build_type=Release, targets=x86).
- **Mathematikbibliothek**: openblas und intel-oneapi-mkl mit entsprechenden Varianten, wobei bei intel-oneapi-mkl die Variante mpi_family immer automatisch an die gewählte MPI-Implementierung angepasst wird.

Zusätzlich existiert COMPILER_CFLAGS_SPACE, der neun vordefinierte Compiler-Flag-Kombinationen enthält: Optimierungsstufen (-O2, -O3), architekturspezifische Flags (-march=native), numerisch aggressive Flags (-ffast-math), Positionsunabhängiger Code (-fPIC) sowie Präprozessor-Definitionen (-DNDEBUG). Eine Konfiguration wird durch die Klasse Configuration repräsentiert, die MPI-Implementierung, MPI-Flags, Compiler, Compiler-Flags, Compiler-C-Flags, Mathematikbibliothek und Math-Flags enthält. Jede Konfiguration erhält eine einzigartige ID (UID), die als die ersten 16 Zeichen des SHA-256-Hashes der JSON-serialisierten Konfiguration berechnet wird und gleichzeitig als Verzeichnisname für das zugehörige Spack-Environment dient. Nicht alle Kombinationen aus Paketen und Varianten sind gültig. In config_proposer.py sind daher vier Arten von Constraints definiert, die in validate_config() geprüft werden:

- CONDITIONAL_FLAGS: legt fest, dass bestimmte Varianten nur gesetzt sein dürfen, wenn eine andere Variante einen bestimmten Wert hat – beispielsweise darf flang=True bei LLVM nur gesetzt werden, wenn auch clang=True gilt.
- MUTEX_VARIANTS: beschreibt sich gegenseitig ausschließende Flag-Gruppen; bei LLVM können etwa lld und gold nicht gleichzeitig aktiv sein.
- CROSS_INCOMPATIBLE und TRIPLE_INCOMPATIBLE: beschreiben paketübergreifende Inkompatibilitäten. So ist mvapich@4.0 nicht mit intel-oneapi-compilers kombinierbar und die Kombination aus openmpi, openblas und llvm lässt sich vom Spack-Concretizer nicht auflösen.

5.2 Neue Konfigurationen

Die Erzeugung neuer Konfigurationen erfolgt in config_proposer.py durch die Funktion propose_next(). Sie implementiert eine Explore-Exploit-Strategie abhängig von der Anzahl bisher erfolgreicher Evaluierungen:

- Solange weniger als EXPLOIT_AFTER (standardmäßig 5) erfolgreiche Evaluierungen vorliegen, wird ausschließlich exploriert: Konfigurationen werden durch random_config() vollständig zufällig aus dem Suchraum gesampelt.

- Sobald genügend erfolgreiche Evaluierungen vorliegen, wechselt das Programm in die Exploitation-Phase, in der neue Konfigurationen durch Mutation bereits bekannter guter Konfigurationen erzeugt werden (exploit_config()).

Bereits gesehene Konfigurationen werden übersprungen. Falls nach max_retries (50) Versuchen keine neue, gültige Konfiguration gefunden wird, fällt das Programm auf rein zufälliges Sampling mit bis zu 500 weiteren Versuchen zurück. Schlägt auch das fehl, wird ein RuntimeError ausgelöst. random_config() wählt für jede der drei Dimensionen zufällig eine Implementierung und ruft sample_flags() auf, das mit einer Einschlusswahrscheinlichkeit von 75% für jedes Flag einen zufälligen Wert aus dem jeweiligen Wertebereich zieht. Die Idee war, dass dadurch vermehrt die Default-Werte genommen werden. Die Compiler-C-Flags werden durch sample_cflags() als zufällig gewählte Kombination aus COMPILER_CFLAGS_SPACE bestimmt. Die so erzeugte Konfiguration wird anschließend durch validate_config() geprüft; ungültige Konfigurationen führen zu einem erneuten Versuch. In der Exploitation-Phase wird zunächst eine Elternkonfiguration aus den erfolgreichen Evaluierungen ausgewählt. Die Auswahlwahrscheinlichkeit ist proportional zur Güte und wird über eine Softmax-Funktion berechnet:

$$w_i = \frac{e^{-s_i/T}}{\sum_j e^{-s_j/T}}$$

wobei s_i der Score der i-ten Konfiguration und T die Temperatur (standardmäßig 0,5) ist. Da ein niedrigerer Score einer besseren Konfiguration entspricht, bewirkt die Negation, dass leistungsstärkere Konfigurationen mit höherer Wahrscheinlichkeit als Eltern ausgewählt werden.

Listing 5.1: Softmax-Gewichtung für die Elternauswahl

```
1
2  def softmax_weights(scores, temperature):
3
4  negated = [-s for s in scores]
5
6  max_val = max(negated)
7
8  exps    = [math.exp((v - max_val) / temperature) for v in negated]
9
10 total   = sum(exps)
11
12 return [e / total for e in exps]
```

Nach der Elternauswahl wird die neue Konfiguration durch Mutation erzeugt. Jede der drei Paketdimensionen wird mit einer Wahrscheinlichkeit von 25% durch eine zufällig gewählte Alternative ersetzt; in diesem Fall werden die Flags der neuen Dimension vollständig neu gesampelt. Bleibt die Dimension erhalten, werden einzelne Flags durch mutate_flags() mutiert: Nur Flags mit mehr als einem möglichen Wert können geändert

werden und der aktuelle Wert wird durch einen anderen zulässigen Wert aus dem Wertebereich ersetzt. Die Compiler-C-Flags werden durch mutate_cflags() ebenfalls mit 25% Wahrscheinlichkeit durch eine andere Kombination aus COMPILER_CFLAGS_SPACE ersetzt. Abschließend korrigiert fix_mkl_mpi_family() bei Bedarf die mpi_family-Variante von intel-oneapi-mkl: mvapich@4.0 wird dabei auf mpich abgebildet.

5.3 Optimierungsschleife

Die übergeordnete Steuerung liegt in optimizer.py in der Funktion run_optimization(). Zu Beginn wird ein vorhandener Checkpoint geladen, der alle bisherigen Evaluierungen als Liste von EvalResult-Objekten enthält. Anschließend folgt eine Startup-Phase: Zunächst werden bereits bekannte Environments aus dem startup_root-Verzeichnis geladen und evaluiert, ohne sie neu bauen zu müssen. Die verbleibenden Plätze bis zur konfigurierten startup_length (standardmäßig 30) werden durch rein zufällige Konfigurationen gefüllt, indem propose_next() mit einem sehr hohen exploit_after-Wert aufgerufen wird, sodass ausschließlich exploriert wird. Nach der Startup-Phase folgt die eigentliche Optimierungsschleife über das konfigurierte Budget (standardmäßig 30 Iterationen). In jeder Iteration wird über propose_next(), nun mit den tatsächlichen Parametern für Mutationsrate, exploit_after und Temperatur, eine neue Konfiguration erzeugt, evaluiert und das Ergebnis dem Checkpoint hinzugefügt. Der Checkpoint wird nach jeder Evaluierung gespeichert, sodass ein Programmabbruch keine abgeschlossenen Evaluierungen verloren gehen lässt.

5.4 Generierung von Environments

Die Übersetzung einer Configuration in ein installiertes Spack-Environment übernimmt config_builder.py. Da alle drei unterstützten Compiler (gcc, llvm, intel-oneapi-compilers) selbst durch Spack gebaut werden müssen, bevor sie als Compiler für MPI und Mathematikbibliotheken verwendet werden können, läuft der Build stets über

_build_config_bootstrap() in drei Phasen ab. In Phase 1 wird ein frisches Spack- Environment angelegt und der Compiler als einziges Paket hinzugefügt. Dann wird ein SLURM-Batch-Skript (build.sh) ins Environment-Verzeichnis geschrieben und über sbatch eingereicht. Das Skript führt auf einem Knoten der Partition west spack concretize und spack install aus und schreibt abschließend SUCCESS oder FAILED in eine Statusdatei. Das Hauptprogramm wartet aktiv auf den Abschluss, indem es squeue periodisch abfragt. In Phase 2 wird nach erfolgreichem Compiler-Build dessen Installationspfad und Versionsnummer über spack find –json und spack location ermittelt. Anschließend wird eine compilers.yaml ins Environment- Verzeichnis geschrieben, die den neu installierten Compiler mit den korrekten Binärpfaden für C, C++ und Fortran registriert. Für gcc sind das gcc/g++/gfortran, für llvm clang/clang++/flang und für intel-oneapi-

compilers icx/icpx/ifx. In Phase 3 werden MPI-Bibliothek und Mathematikbibliothek mit dem in Phase 2 registrierten Compiler zum Environment hinzugefügt und erneut über einen SLURM-Batch-Job konkretisiert und installiert. Erfolgreich gebaute Konfigurationen können optional in einem startup_root-Verzeichnis gespeichert werden, sodass sie beim nächsten Programmstart ohne erneuten Build direkt evaluiert werden können.

5.5 Evaluierung

Der Einstiegspunkt der Evaluierung ist die Funktion evaluate() in config_evaluator.py. Sie ruft nacheinander build_config(), build_spack_benchmarks(), run_spack_benchmarks() und wait_and_score() auf und bricht bei einem Fehler in jedem Schritt frühzeitig ab. Die Ergebnisse werden in einer EvalResult-Instanz gesammelt, die neben dem kombinierten Score die Einzelwerte benchmark_hpl und benchmark_osu, die Build-Dauer, den rohen Log-Output und ein Dictionary für zusätzliche Metriken enthält. Schlägt ein Schritt fehl, wird error_msg gesetzt und success bleibt False. build_spack_benchmarks() fügt dem bestehenden Spack-Environment osu-micro-benchmarks und hpl als weitere Pakete hinzu und installiert sie über einen SLURM-Batch-Job. Eigentlich sollte auch hpcc in diesem Schritt installiert und im späteren Verlauf ausgeführt werden. Es gelang jedoch nicht, HPCC erfolgreich zu installieren und auszuführen. Nach erfolgreichem Build wird die Eingabedatei HPL.dat mit den Benchmark-Parametern ins Environment-Verzeichnis geschrieben. Die Ausführung der Benchmarks erfolgt in run_spack_benchmarks() durch Einreichen eines SLURM-Batch-Skripts (bench.sh), das auf zwei Knoten mit insgesamt 24 Tasks (12 pro Knoten) auf der Partition west mit einem Zeitlimit von 45 Minuten läuft. Das Skript aktiviert das Spack-Environment und führt drei Gruppen von Benchmarks aus:

- **HPL**: xhpl mit 12 MPI-Prozessen.
- **OSU Punkt-zu-Punkt**: osu_latency, osu_bw und osu_bibw mit je 2 Prozessen (100 Warm-up-Iterationen, 1000 Messiterationen).
- **OSU One-Sided**: osu_put_latency, osu_get_latency, osu_put_bw, osu_get_bw und osu_acc_latency mit je 2 Prozessen.
- **OSU Kollektiv**: osu_allreduce, osu_bcast, osu_barrier, osu_allgather, osu_reduce, osu_scatter und osu_gather mit allen 24 Prozessen.

wait_and_score() wartet mit einem Timeout von 7200 Sekunden auf den Abschluss des Jobs und parst anschließend die Ausgabedateien. Aus den Ergebnissen wird ein kombinierter Score berechnet, der bewusst einfach gehalten wurde:

$$\text{score} = \frac{\text{osu\_latency}}{\text{hpl\_gflops} \cdot \text{osu\_bw}}$$

Dabei fließen HPL-Gflops aus der WR-Zeile der HPL-Ausgabe, die 1-Byte-Latenz aus osu_latency und die maximale Bandbreite über alle Nachrichtengrößen aus osu_bw ein. Ein niedriger Score steht für eine gute Konfiguration: hohe Rechenleistung und Netzwerkbandbreite im Nenner, geringe Latenz im Zähler. Nur wenn alle drei Messwerte gültig sind, gilt die Evaluierung als erfolgreich.

6 Auswertung

Das Installieren und Testen aller möglichen Environments war aufgrund der großen Anzahl an Variantenkombinationen nicht realisierbar. Insgesamt ergeben sich 1.579.776.000 mögliche Environments.

Ein Großteil der erzeugten Environments lieferte keine verwertbaren Ergebnisse. Dies hatte verschiedene Ursachen. Zum einen konnten viele Environments entgegen den Erwartungen nicht konkretisiert oder anschließend nicht installiert werden. Ursprünglich wurde erwartet, dass Spack einen Großteil dieser Probleme automatisch behandelt und dadurch eine manuelle Einschränkung der Variants weitgehend überflüssig macht. Zudem waren die ausgegebenen Fehlermeldungen bei Installationsfehlern häufig wenig aussagekräftig, wodurch die Fehlersuche erschwert wurde.

Zum anderen traten Probleme bei der Installation und Ausführung der Benchmarks auf.

Bei HPL kam es beispielsweise vor, dass Benchmarks nach einem Fehler nur noch ohne Batch-Skript auf dem Login-Knoten ausgeführt werden konnten. Zu einem späteren Zeitpunkt war die Ausführung auf den Rechenknoten jedoch wieder möglich.

Ein weiteres Problem betraf OpenMPI. Hier funktionierte mpirun nur mit aktivierter Variant legacylauncher. (Außer in einem Fall. Ich habe keine Ahnung wo der Benchmark herkommt oder warum er funktioniert hat.) Diese Variant wurde bei der ursprünglichen Auswahl ausgeschlossen, da angenommen wurde, dass es sich lediglich um ein veraltetes Feature handelt. Erst zu einem späten Zeitpunkt wurde festgestellt, dass legacylauncher nur bis OpenMPI Version 4 unterstützt wird. Aus diesem Grund liegen für OpenMPI nur sehr wenige Messergebnisse vor.

Mit MPICH traten vergleichsweise wenige Probleme auf. Einige Variantenkombinationen waren zwar nicht funktionsfähig, dies war jedoch bereits erwartet worden.

Die größten Schwierigkeiten verursachte MVAPICH. Keines der erzeugten Environments konnte erfolgreich installiert werden.

Die durch LLVM verursachten Fehler konnten ebenfalls nicht behoben werden, weshalb für diesen Compiler keine Ergebnisse vorliegen. Die beiden anderen Compiler funktionierten dagegen weitgehend problemlos.

Mit den verschiedenen BLAS-Implementierungen traten nur wenige Schwierigkeiten auf. Auch die getesteten Variants erwiesen sich insgesamt als vergleichsweise unproblematisch.

Dadurch verblieben von den ursprünglich vorgesehenen 18 Paketkombinationen lediglich die in Tabelle 6.1 dargestellten Kombinationen. Da ich aber nie die Möglichkeit

Compiler	MPI	BLAS
gcc	mpich	openblas
gcc	openmpi	openblas
gcc	mpich	mkl
icc	mpich	mkl
icc	mpich	openblas

Tabelle 6.1: Stacks mit Ergebnissen

ausgeschaltet habe, dass sie in anderen Kombination genutzt werden, sind alle Variationen theoretisch immer noch möglich. Die Variants habe ich mit der Zeit auch immer weiter eingeschränkt bis nur noch 32 übrig waren (siehe Tabelle 6.2). Damit sind nur noch 1.105.920 Kombinationen möglich und mit den Compilerflags sind es dann 11.059.200 mögliche Environments.

Ich habe erwartet, dass die hohe Anzahl an Versuchen die Auswirkungen einzelner Probleme reduziert. Für viele Variants liegen nur wenige Messwerte vor und häufig wurden gleichzeitig weitere Variants oder Compiler-Flags verändert. Dadurch ist eine eindeutige Zuordnung beobachteter Leistungsunterschiede zu einzelnen Einflussfaktoren nur eingeschränkt möglich und die Resultate mit Vorsicht zu interpretieren. Die Ergebnisdaten liegen in [git26].

6.1 Cluster und Parameter

Der zum Bauen und Testen der Environments verwendete Rechencluster ist der Studentencluster des DKRZ. Die Rechenknoten verfügen über zwei x86_64-Prozessoren mit jeweils sechs physischen Kernen sowie 11 GB Arbeitsspeicher. Die Knoten sind über Ethernet miteinander verbunden.

Auf dem Cluster ist Slurm-WLM 19.05.5 installiert. Für Spack steht standardmäßig die Version 0.19.0.dev0 zur Verfügung. Für diese Arbeit wurde jedoch eine eigene Spack-Installation in Version 1.1.1 eingerichtet und für sämtliche Experimente verwendet.

Für die HPL-Benchmarks wurden die in [adv17] beschriebenen Parameter als Ausgangspunkt verwendet. Zunächst wurde versucht, den gesamten verfügbaren Arbeitsspeicher von 11 GB zu nutzen. Da dies wiederholt zu Programmabbrüchen führte, wurde die Speichernutzung schrittweise reduziert. Getestet wurden 11 GB, 8 GB, 6 GB, 4 GB und 3 GB. Auch bei 4 GB traten vereinzelt Abstürze auf, weshalb für die weiteren Experimente eine Speichergröße von 3 GB gewählt wurde. Die Anzahl der verwendeten Knoten wurde auf zwei festgelegt, wobei pro Knoten sechs Kerne genutzt wurden.

Um Einflüsse durch Kaltstarts bei den OSU Micro-Benchmarks zu vermeiden, wurden vor jeder Messung 1000 Aufwärmläufe durchgeführt. Anschließend wurden weitere 100 Messläufe ausgeführt und deren Median als Ergebnis verwendet.

Paket	Variant	Werte
mpich	hwloc	True, False
mpich	pci	True, False
mpich	romio	True, False
mpich	vci	True, False
mpich	wrapperrpath	True, False
mpich	netmod	ofi, ucx
mpich	pmi	pmi2, pmix
openmpi	atomics	True, False
openmpi	fortran	True, False
openmpi	pmi	True, False
openmpi	romio	True, False
openmpi	rsh	True, False
openmpi	vt	True, False
openmpi	wrapper-rpath	True, False
openmpi	fabrics	ucx, ofi
openmpi	schedulers	slurm, none
gcc	binutils	True, False
gcc	bootstrap	True, False
gcc	graphite	True, False
gcc	strip	True, False
icc	envmods	True, False
openblas	dynamic_dispatch	True, False
openblas	fortran	True, False
openblas	locking	True, False
openblas	pic	True, False
openblas	threads	openmp, none
intel-oneapi-mkl	cluster	False, True
intel-oneapi-mkl	envmods	False, True
intel-oneapi-mkl	gfortran	False, True
intel-oneapi-mkl	mpi_family	openmpi, mpich
intel-oneapi-mkl	shared	False, True
intel-oneapi-mkl	threads	openmp, none

Tabelle 6.2: finale Variantsauswahl

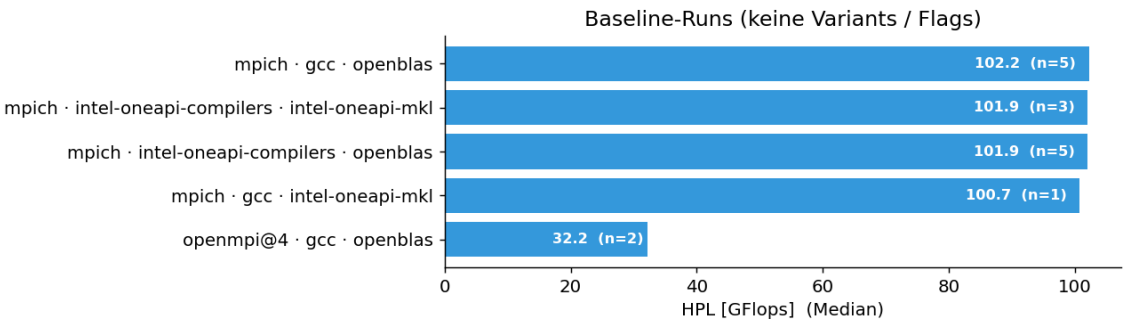

Abbildung 6.1: HPL Baseline

6.2 Resultate

6.2.1 Software-Stack

Um eine Baseline für die weiteren Auswertungen zu erhalten, wurden zunächst Benchmarks ohne zusätzliche Variants oder Compiler-Flags durchgeführt. Die Ergebnisse für HPL und die OSU Micro-Benchmarks sind in Abbildung 6.1 beziehungsweise Abbildung 6.2 dargestellt.

Die HPL-Ergebnisse liegen sehr nahe beieinander und zeigen nur geringe Unterschiede zwischen den getesteten Software-Stacks. Die einzige Ausnahme bietet openmpi-gcc-openblas, welches nur ca. 32 % der anderen Software-Stacks erreicht.

Die OSU-Benchmarks weisen dagegen deutliche Abweichungen auf, die über mehrere Benchmarkdurchläufe hinweg konsistent beobachtet wurden. Besonders fällt auf, dass OpenMPI eine deutlich geringere Latenz und eine deutlich höhere Bandbreite aufweist als die anderen Software-Stacks.

6.2.2 Compiler

Der Intel-Compiler erzielte im HPL-Benchmark gute Performance. Auch wenn es in Abbildung 6.3 aussieht als würde gcc bei HPL sehr schlecht performen, gibt es ein paar Benchmarks in denen gcc deutlich besser performt. Wenn die Compiler-Flags hinzukommen, zeigt sich bei gcc ein starker Leistungsabfall (Abbildung 6.4). Wenn man sich die Daten etwas genauer anschaut, kann man feststellen, dass gcc sowohl sehr gute HPL-Benchmarks als auch sehr schlechte hat.

6.2.3 Variants

Boolean-Variants

Die Variants von MPI und BLAS waren leider nicht so spannend. In allen Fällen war der Standardwert auch der im Durchschnitt der beste Wert für HPL (Abbildung 6.5). Bei den OSU-Benchmarks ergab sich ein ähnliches Bild (Abbildung 6.6). Viele der Nicht-

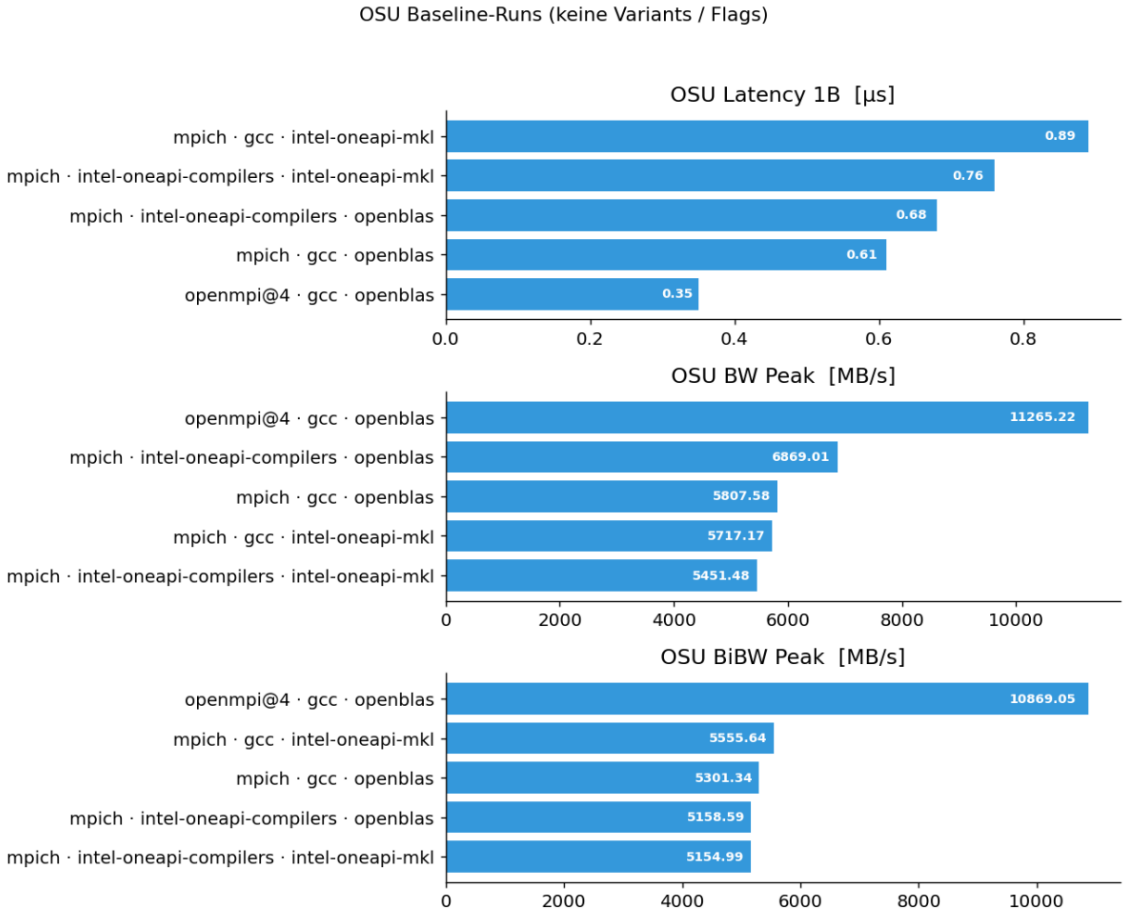

Abbildung 6.2: OSU Baseline

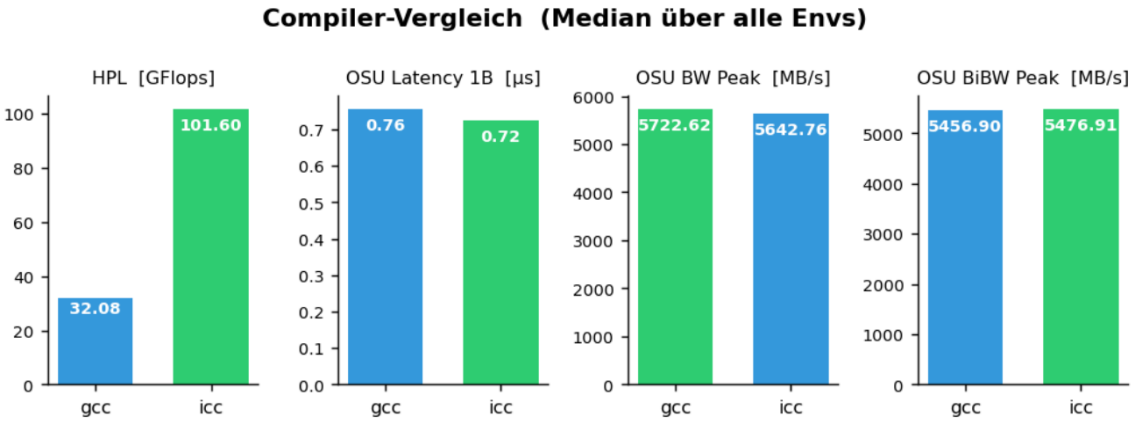

Abbildung 6.3: Compiler Median über alle Env

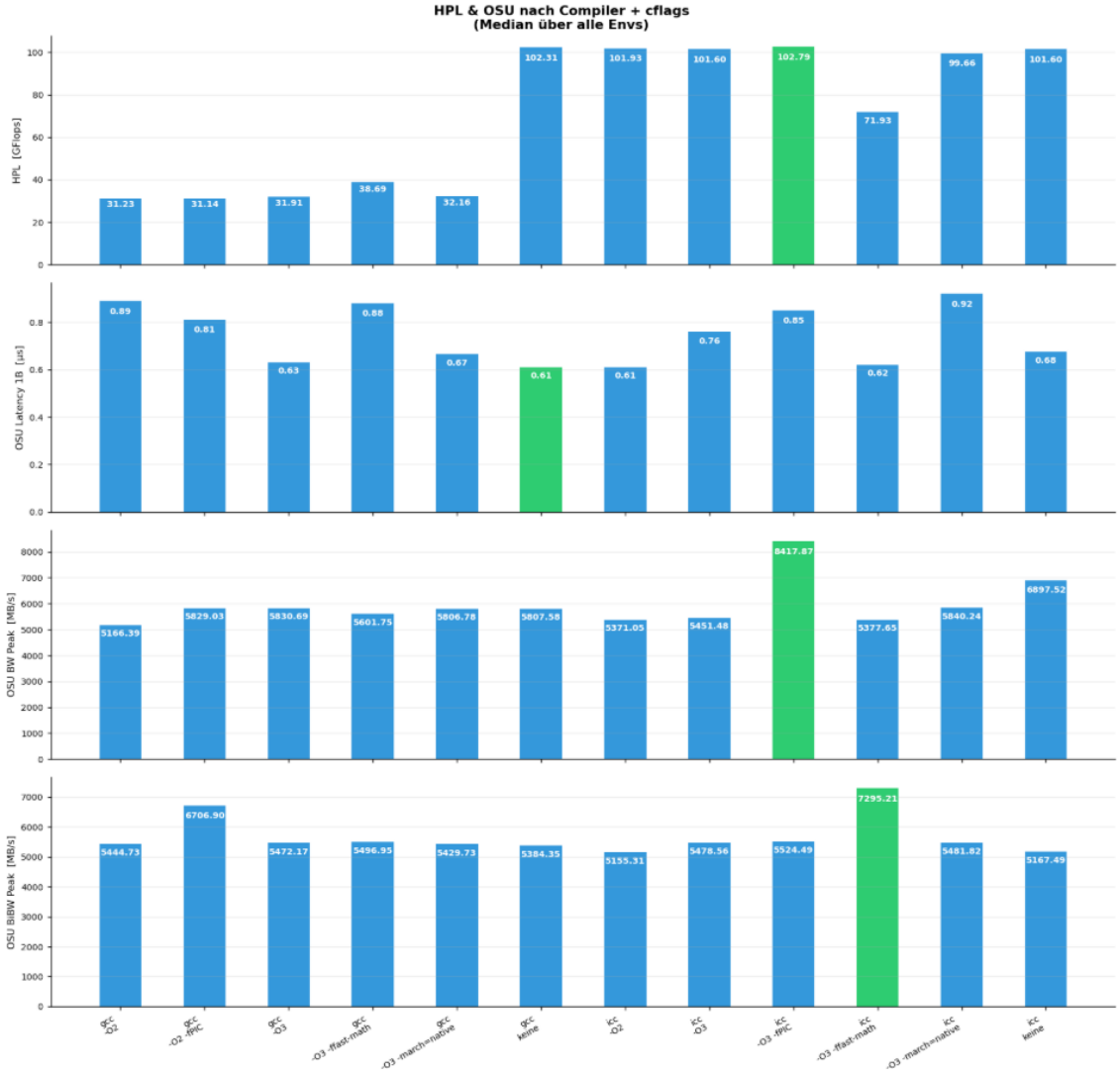

Abbildung 6.4: HPL & OSU Benchmarks Flags

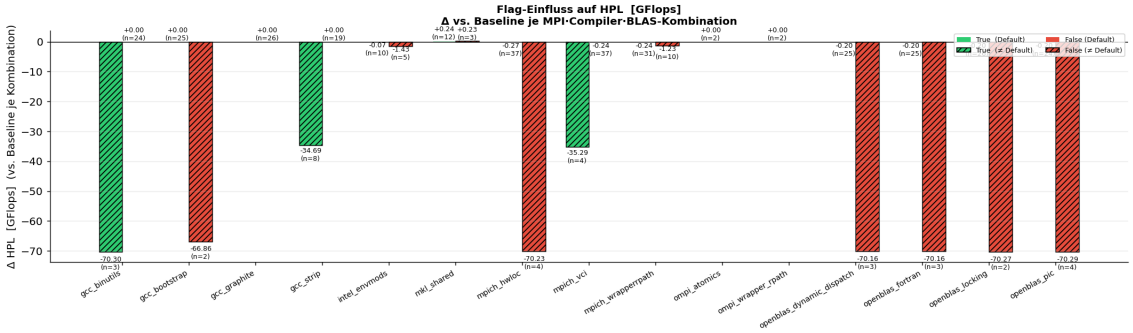

Abbildung 6.5: HPL - Boolean-Variants

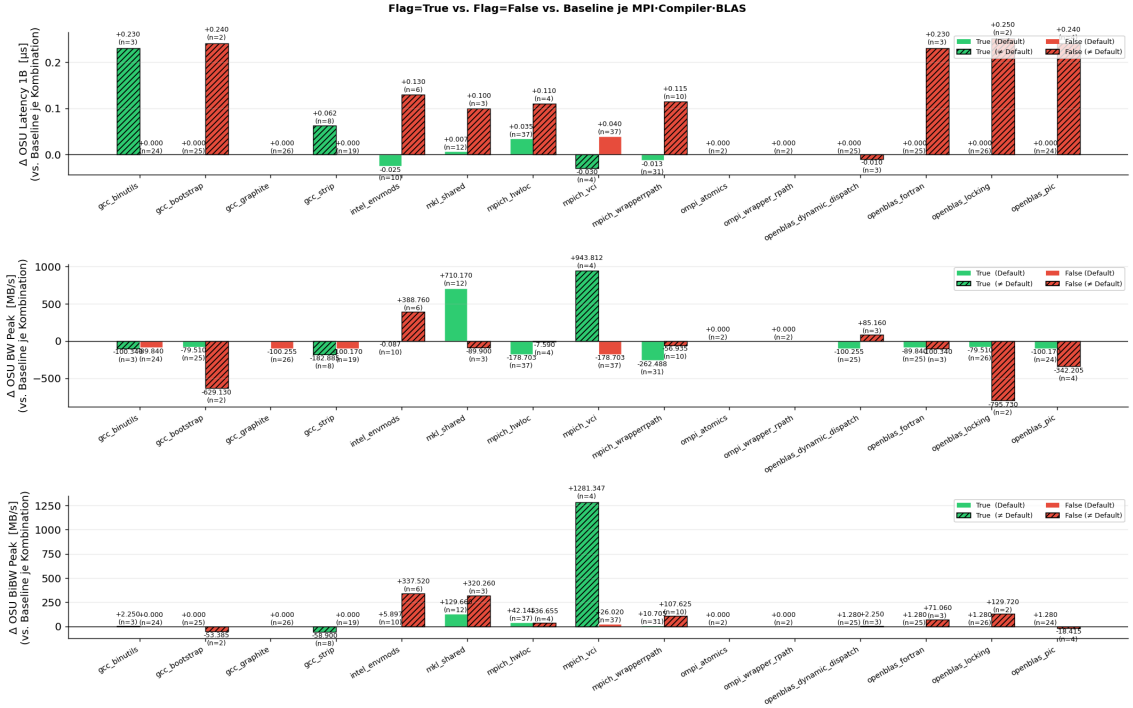

Abbildung 6.6: OSU - Boolean-Variants

Standardwerte hatten eine schlechter Latenz. Bei der Bandbreite gab es wenige signifikante Ausschläge.

Die einzige Ausnahme ist vci (Virtual Communication Interface). Es ist per Default aus, verringert aber, wenn es aktiviert ist, die Latenz minimal um 0,03 µs verringert und es erhöht den Durchsatz in eine Richtung um 940 MB/s und beidseitig um 1280 MB/s im Vergleich zur Baseline. Die Aktivierung bringt aber gleichzeitig einen Rechenoverhead mit, den man in der HPL-Benchmark sehen kann (-35 GFLOPS).

Multi-Variants

Auch hier war der Standardwert in der HPL-Benchmark (fast) immer der Beste (Abbildung 6.7). Allerdings hatten nur drei der Variants einen signifikanten Einfluss.

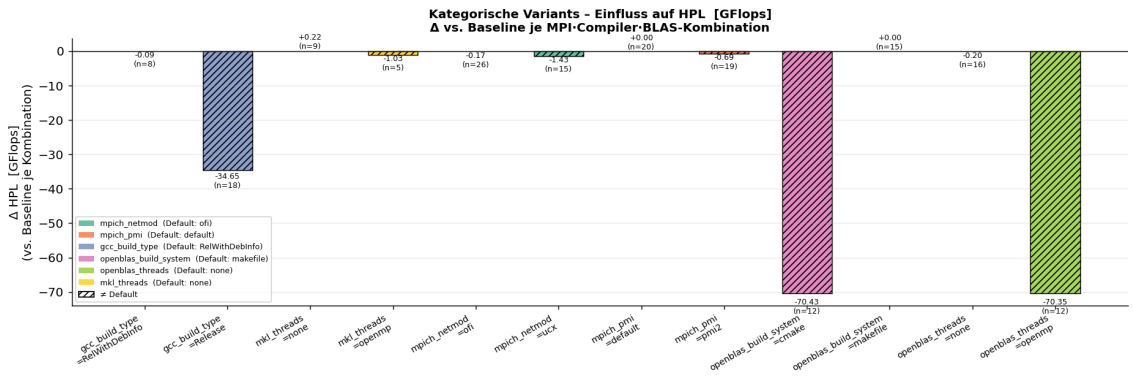

Abbildung 6.7: HPL - Multi-Variants

In den OSU-Benchmarks (Abbildung 6.8) gab es bei der Latenz keine positiven Effekte im Vergleich mit den Standardwerten. Bei den einseitigen Bandbreitenmessungen hatten nur mkl's „threads=openmp"einen positiven Einfluss. Die bidirektionalen Messungen ergaben, dass die Standardwerte im Vergleich etwas schlechter abschnitten.

6.3 Diskussion der Ergebnisse

Die Ergebnisse sind insgesamt nicht ausreichend, um ein vollständiges Bild davon zu zeichnen, welche Variants den größten Einfluss auf die Performance haben. Dafür sind zu viele Kombinationen ungetestet geblieben und zu viele Messreihen wurden durch technische Probleme – insbesondere mit OpenMPI und MVAPICH – verworfen oder konnten nie vollständig abgeschlossen werden.

Dennoch lassen sich einige Tendenzen erkennen. Beim Software-Stack zeigte sich, dass die Wahl der MPI-Implementierung vor allem für die Kommunikationslatenz und den Durchsatz relevant ist. OpenMPI lieferte in den OSU-Benchmarks deutlich bessere Werte als die MPICH-basierten Stacks. Da der überwiegende Teil der Ergebnisse auf MPICH basiert, war dieses Verhalten unerwartet. Ich hatte angenommen, dass die OSU-Benchmarks weitgehend identische Ergebnisse liefern würden und sich Unterschiede hauptsächlich in den mit HPL gemessenen GFLOPS-Werten zeigen würden. OpenMPI war jedoch wegen des Problems mit legacylauncher nur in sehr wenigen Konfigurationen verwendbar, weshalb die Aussagekraft dieser Ergebnisse begrenzt bleibt. MPICH erwies sich dagegen als deutlich stabiler und zuverlässiger.

Bei den Compilern fiel auf, dass der Intel-Compiler (icc) im HPL-Benchmark konsistent gut abschnitt – was angesichts der verwendeten Intel-CPU zu erwarten war. GCC zeigte ein deutlich durchwachseneres Bild. Das hat mich enorm überrascht. Ich hatte erwartet, dass es eine Performancesteigerung bei den HPL-Benchmarks geben würde aufgrund des zu erwartenden Geschwindigkeitszuwachses. Ohne Compiler-Flags war die Performance vergleichbar, während das Hinzufügen von Optimierungs-Flags, entgegen der Erwartungen, in vielen Fällen zu einem starken Leistungseinbruch führte. Obwohl

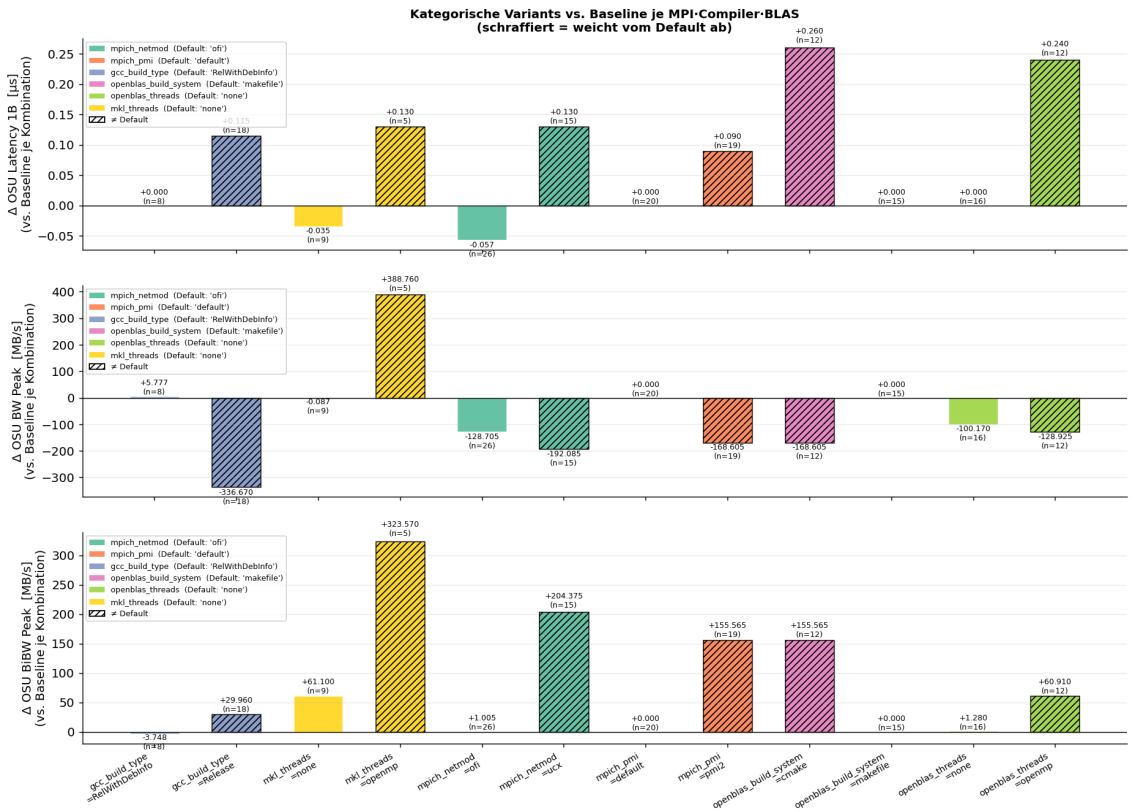

Abbildung 6.8: OSU - Multi-Variants

auch einige der besten Ergebnisse mit gcc-Flags erzielt wurden. Dieser Befund ist kontraintuitiv und lässt sich möglicherweise auf ungünstige Wechselwirkungen zwischen bestimmten Flags und den jeweiligen Variants oder der Laufzeitumgebung zurückführen. Eine genauere Analyse wäre notwendig, um die Ursache zu isolieren.

Bei den untersuchten Variants erwiesen sich in den meisten Fällen die Standardkonfigurationen als die beste Wahl. Ausnahmen bildeten die MKL-Variant „threads=openmp“sowie die MPICH-Variant „vci“. Dass „threads=openmp“nicht standardmäßig aktiviert ist, erscheint nachvollziehbar, da OpenMP nicht auf jedem Zielsystem verfügbar ist. Auf Systemen mit installierter OpenMP-Unterstützung sollte die Aktivierung jedoch in Betracht gezogen werden. Überraschend war, dass die entsprechende Variant bei OpenBLAS keine vergleichbaren Leistungsgewinne zeigte.

Auch die standardmäßig deaktivierte MPICH-Variant „vci“ist nachvollziehbar, da ihre Nutzung an die gleichzeitige Aktivierung von „device=ch4“gebunden ist. Die Ergebnisse deuten darauf hin, dass „vci“insbesondere für kommunikationsintensive Anwendungen Vorteile bietet, während die HPL-Leistung darunter leidet.

Insgesamt zeigt sich, dass die Konfigurationsvarianten zwar prinzipiell messbare Auswirkungen haben können, der Effekt jedoch stark von der restlichen Umgebung abhängt. Die Wechselwirkungen zwischen MPI-Implementierung, Compiler, BLAS und den jeweiligen Variants sind komplex und lassen sich nicht einfach isolieren. Eine systematischere

Untersuchung – mit mehr kontrollierten Experimenten und einem stabileren Testaufbau, wäre nötig, um belastbare Aussagen treffen zu können.

7 Fazit

Ziel dieser Arbeit war es, zu untersuchen, inwiefern sich verschiedene Software-Stack-Konfigurationen, bestehend aus MPI-Implementierung, Compiler, BLAS-Bibliothek und den jeweiligen Spack-Variants, auf die Performance von HPC-Anwendungen auswirken. Dazu wurden mit Spack automatisiert Environments erzeugt und anschließend mit dem HPL-Benchmark sowie den OSU Micro-Benchmarks evaluiert.

Die praktische Durchführung war mit erheblichen Herausforderungen verbunden. Viele der erzeugten Environments ließen sich nicht konkretisieren oder installieren, und technische Probleme, insbesondere mit OpenMPI und MVAPICH, reduzierten die Anzahl auswertbarer Konfigurationen erheblich. Auch die Qualität der Fehlermeldungen von Spack erwies sich als unzureichend für eine effiziente Fehleranalyse.

Trotz dieser Einschränkungen konnten einige Erkenntnisse gewonnen werden. Die Wahl des Software-Stacks hat vor allem auf die Kommunikationsperformance einen deutlichen Einfluss: OpenMPI erzielte signifikant bessere Ergebnisse als MPICH, war jedoch aufgrund von Kompatibilitätsproblemen nur eingeschränkt nutzbar. Beim rechenintensiven HPL-Benchmark zeigten sich zwischen den verbleibenden Stacks hingegen kaum Unterschiede, mit Ausnahme von OpenMPI, das hier stark unterdurchschnittlich abschnitt. Der Intel-Compiler lieferte auf Intel-Hardware erwartungsgemäß gute Ergebnisse, während GCC in Kombination mit bestimmten Optimierungs-Flags unerwartet schlechte Performance zeigte. Die untersuchten Variants hatten insgesamt nur geringen Einfluss auf die Ergebnisse, was das eigentliche Ziel der Arbeit war.

Für zukünftige Arbeiten wäre eine Erweiterung des Testclusters auf unterschiedliche Hardware sowie eine gezieltere Vorauswahl funktionsfähiger Konfigurationen sinnvoll, um den Anteil auswertbarer Messreihen zu erhöhen. Zudem sollten einzelne Einflussfaktoren, insbesondere Compiler-Flags und MPI-Variants, in isolierten Experimenten untersucht werden, um Wechselwirkungseffekte besser zu verstehen. Ein verbessertes Monitoring der Spack-Builds und aussagekräftigere Fehlermeldungen würden die Reproduzierbarkeit und Effizienz solcher Experimente erheblich steigern.

Literaturverzeichnis

[adv17] *How do I tune my HPL.dat file? - Advanced Clustering Technologies*. https://www.advancedclustering.com/act_kb/tune-hpl-dat-file. Version: Februar 2017. – [Online; accessed 15. Jun. 2026]

[Bib24] *OSU Micro-benchmarks - UL HPC Tutorials*. https://ulhpc-tutorials.readthedocs.io/en/latest/parallel/mpi/OSU_MicroBenchmarks. Version: Juli 2024. – [Online; accessed 6. May 2026]

[DLP03] DONGARRA, Jack J. ; LUSZCZEK, Piotr ; PETITET, Antoine: The LINPACK Benchmark: past, present and future. In: *Concurrency and Computation: Practice and Experience* 15 (2003), Nr. 9, 803-820. http://dx.doi.org/https://doi.org/10.1002/cpe.728. – DOI https://doi.org/10.1002/cpe.728

[DS03] DIMITROV, Rossen ; SKJELLUM, Anthony: Software architecture and performance comparison of MPI/Pro and MPICH. In: *International Conference on Computational Science* Springer, 2003, S. 307–315

[DTB+04] DONGARRA, Jack J. ; TAKAHASHI, Daisuke ; BAILEY, David ; KOESTER, David ; LUSZCZEK, Piotr ; RABENSEIFNER, Rolf ; LUCAS, Bob ; MCCALPIN, John: *Introduction to the HPC Challenge benchmark suite*. 2004

[git26] *Avvalinja/BA_Ergebnisse2026*. https://github.com/Avvalinja/BA_Ergebnisse2026/tree/main/Benchmarks_Env. Version: Juni 2026. – [Online; accessed 17. Jun. 2026]

[GLC+15] GAMBLIN, Todd ; LEGENDRE, Matthew ; COLLETTE, Michael R. ; LEE, Gregory L. ; MOODY, Adam ; SUPINSKI, Bronis R. ; FUTRAL, Scott: The Spack package manager: bringing order to HPC software chaos. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA : Association for Computing Machinery, 2015 (SC '15). – ISBN 9781450337236

[GLS99] GROPP, William ; LUSK, Ewing ; SKJELLUM, Anthony: *Using MPI: portable parallel programming with the message-passing interface*. Bd. 1. MIT press, 1999

[GRB+12] In: GRUNE, Dick ; REEUWIJK, Kees van ; BAL, Henri E. ; JACOBS, Ceriel J. H. ; LANGENDOEN, Koen: *Interpretation*. New York, NY : Springer New York, 2012. – ISBN 978–1–4614–4699–6, 299–312

[PDMS+21] POENARU, Andrei ; DEAKIN, Tom ; MCINTOSH-SMITH, Simon ; HAMMOND, Simon D. ; YOUNGE, Andrew J.: An evaluation of the fujitsu a64fx for hpc applications. In: *Cray User Group* (2021)

[SK23] SCHRÖDER, Fabian ; KUHN, Michael: Automated performance analysis of software stacks for distributed systems. (2023)

[slu26] *Slurm Workload Manager - Quick Start User Guide*. `https://slurm.schedmd.com/quickstart.html`. Version: Juni 2026. – [Online; accessed 11. Jun. 2026]

[Spa26] *Spec Syntax - Spack 1.2.0.dev0 documentation*. `https://spack.readthedocs.io/en/latest/spec_syntax.html`. Version: Januar 2026. – [Online; accessed 6. May 2026]

[Ste02] STERLING, Thomas L.: *Beowulf cluster computing with Linux*. MIT press, 2002. – 1–19 S.

[YJG03] YOO, Andy B. ; JETTE, Morris A. ; GRONDONA, Mark: Slurm: Simple linux utility for resource management. In: *Workshop on job scheduling strategies for parallel processing* Springer, 2003, S. 44–60

[Yua06] YUAN, WAN: Performance Comparison of Public Domain MPI Implementations using Application Benchmarks. (2006)

[ZH23] ZHU, Mingxuan ; HAO, Dan: Compiler Auto-Tuning via Critical Flag Selection. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, S. 1000–1011

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe. Sofern im Zuge der Erstellung der vorliegenden Abschlussarbeit generative Künstliche Intelligenz (gKI) basierte elektronische Hilfsmittel verwendet wurden, versichere ich, dass meine eigene Leistung im Vordergrund stand und dass eine vollständige Dokumentation aller verwendeten Hilfsmittel gemäß der Guten Wissenschaftlichen Praxis vorliegt. Ich trage die Verantwortung für eventuell durch die gKI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.

Hamburg, den 17.6.2026 Unterschrift: