FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Bachelorarbeit

Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien anhand von verschiedenen Benchmarks auf HPC Systemen

Fachbereich Medizintechnik und Technomathematik
im Studiengang Angewandte Mathematik und Informatik

Jülich, den 22. September 2021

Autor: Julia Wellmann (Matr. Nr.: 3209301)
1. Prüfer: Prof. Dr.-Ing. Andreas Terstegge
2. Prüfer: Sebastian Lührs

Firma: Forschungszentrum Jülich GmbH
Institut: Jülich Supercomputing Centre

JÜLICH
Forschungszentrum

Eidesstattliche Erklärung

Hiermit versichere ich, Julia Wellmann, dass ich die Bachelorarbeit mit dem Thema

Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien anhand von verschiedenen Benchmarks auf HPC Systemen

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.

Ort und Datum

Unterschrift des Autors

Vorwort

Die vorliegende Bachelorarbeit beschäftigt sich mit der Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien anhand von verschiedenen Benchmarks auf HPC Systemen. Diese Arbeit verfasste ich als Abschlussarbeit meines dualen Studiums der angewandten Mathematik und Informatik B.Sc. an der Fachhochschule Aaachen im Rahmen meiner Tätigkeit im Jülich Supercomputing Centre des Forschungszentrum Jülich. Ziel dieser Arbeit war es dabei vornehmlich den Leistungsunterschied der verschiedenen Konfigurationen zu bewerten.

Zusammen mit meinem Betreuer, Sebastian Lührs, entwickelte ich die Fragestellung für diese Bachelorarbeit, basierend auf einer ebenfalls gemeinsam erstellten Fragestellung meiner Seminararbeit ***Entwicklung einer webbasierten Visualisierung des Prozess-Pinnings auf HPC Systemen***[14]. In dieser Seminararbeit wurden die Grundzüge des Prozess-Pinnings und die Entwicklung eines Pinning-Webtools herausgearbeitet. In der vorliegenden Bachelorarbeit sind einige Thematiken der Seminararbeit mit eingegangen, sodass in einzelnen Kapiteln dieser Arbeit auf die Seminararbeit verwiesen wird. Außerdem wurde das erstellte Pinning-Webtool zur Veranschaulichung der Prozess-Pinning-Optionen genutzt.

Julia Wellmann
Jülich, 22. September 2021

Inhaltsverzeichnis

1. Einleitung

In diesem Kapitel wird die Motivation der Arbeit und die Grundlagen zu Hochleistungsrechner Systemen erläutert. In den folgenden Kapiteln 2 und 3 wird die Theorie des Prozess-Pinnings und die verwendeten Softwarepakete beschrieben. Die Kapitel 4 und 5 bilden den Hauptteil der Arbeit und erläutern die Durchführung der Benchmarks und den eigentlichen Leistungsvergleich. Abschließend wird eine Zusammenfassung und einen Ausblick über die Thematik in Kapitel 6 gegeben.

1.1. Motivation

Die Verwendung von Hochleistungsrechner (High Performance Computing (HPC)) Systemen zum effizienten und schnellen Einsatz von entsprechenden Anwendungsprogrammen hat in vielen Feldern der Wissenschaft stark an Bedeutung gewonnen.

Auch das Jülich Supercomputing Centre (JSC) des Forschungszentrum Jülich bietet Wissenschaftlern Zugang und Rechenzeit an seinen Hochleistungsrechnern an. Diese Systeme sind äußerst komplex und bieten eine Vielzahl an Möglichkeiten zur Verwaltung der Ressourcen und der auszuführenden Jobs. Durch die Komplexität ist es oftmals erforderlich, die Nutzer bei der Benutzung dieser Systeme zu unterstützen. Die Unterstützung zielt dabei auf den effizienten Einsatz mehrerer Rechenknoten und des zugrundeliegenden Netzwerks, sowie der effizienten Nutzung jedes einzelnen Rechenknotens ab. Ein relevanter Aspekt zur effizienten Nutzung eines einzelnen Rechenknotens stellt das Prozess-Pinning dar.

Bei den auszuführenden Anwendungsprogrammen handelt es sich in der Regel um Quellcode, der z.B. mittels Message Passing Interface[1] (MPI) oder Open Multi-Processing[2] (OpenMP) parallelisiert ist. Zur parallelen Ausführung wird das Programm mit mehreren Prozessen (Tasks) gestartet, die wiederum aus einem oder mehreren 'Teilprozessen', sogenannten Threads, bestehen. Die Verteilung der Prozesse auf die Prozessorkerne des Rechners, das sogenannte Prozess-Pinning, kann die Leistung der Anwendungsprogramme stark beeinflussen. Der Nutzer kann unter anderem durch verschiedene Optionen des Workload Managers, der die Funktionen eines Schedulers und eines Ressourcenmanagers kombiniert (am JSC kommt eine angepasste Form des Slurm Workload Managers zum Einsatz, siehe Kapitel 2.2), die Prozess-Pinning Strategie und somit die Leistung eines Anwendungsprogramm beeinflussen.

Der Schwerpunkt dieser Arbeit liegt in der Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien auf HPC Systemen. Dafür wurden zwei schon existierende Benchmarks auf dem HPC System JUSUF[3] des JSCs automatisiert ausgeführt und bei jedem Lauf sowohl die Verteilung der Prozesse als auch die Anzahl der Tasks und Threads angepasst, um verschiedene Strategien vergleichen zu können.

[1] https://www.mcs.anl.gov/research/projects/mpi/
[2] https://www.openmp.org/
[3] https://apps.fz-juelich.de/jsc/hps/jusuf/index.html

1.2. Hochleistungsrechensysteme

Das Hochleistungsrechnen beschreibt die Nutzung von parallelen Datenverarbeitungen zur effizienten und schnellen Bearbeitung anspruchsvoller Anwendungsprogramme.

Das JSC bietet derzeit Zugang und Rechenzeit zu verschiedenen HPC Systemen für Forschungsprojekte aus Hochschulen, Forschungsorganisationen und der Industrie an. Jedes dieser HPC Systeme hat eine Anzahl an verfügbaren Rechenknoten, die mittels eines Hochgeschwindigkeitsnetzwerkes miteinander verbunden sind. Zusätzlich zum Zusammenschalten mehrerer Rechenknoten durch eine sehr schnelle Netzwerkinfrastruktur macht die Ausnutzung von mehreren Prozessen auf einem Knoten ein HPC System aus. Ein Knoten besteht aus einem oder mehreren Sockets mit jeweils einem Prozessor (CPU). Jeder Prozessor besteht aus einer festen Anzahl physischer Kerne. Jeder physischer Kern kann mehrere virtuelle Kerne beinhalten (sogenanntes 'Simultaneous Multithreading' (SMT)), die das parallele Ausführen eines Prozesses durch weitere Registersätze ermöglichen.

Jeder Prozessor verfügt über einen mehrstufigen Puffer-Speicher, auch Level-1- (L1-), Level-2- (L2-) und Level-3-Cache (L3-Cache) genannt. Jedem physischen Prozessorkern ist ein eigener L1-Cache und L2-Cache zugeordnet. Alle drei Caches erlauben das Zwischenspeichern der Daten des regulären Arbeitsspeichers und haben als Ziel möglichst selten auf diesen zuzugreifen. Sie unterscheiden sich durch ihre Größe und Zugriffsgeschwindigkeit.

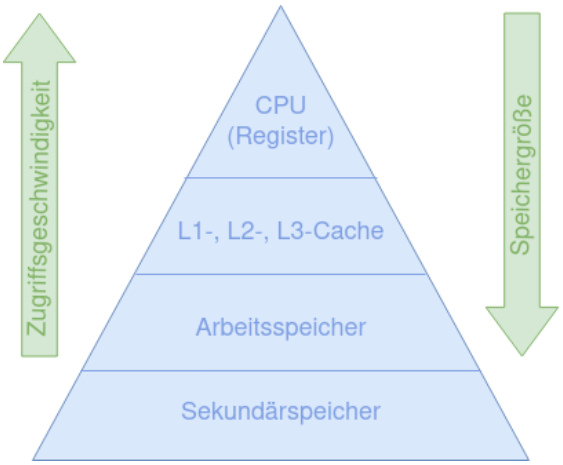


Abbildung 1.1.: Speicherlevel eines Prozessors

Abbildung 1.1 veranschaulicht die abnehmende Zugriffsgeschwindigkeit bei zunehmender Speichergröße der verschiedenen Speicherebenen.

Der Arbeitsspeicher selbst wird oftmals in mehrere Speicherknoten aufgeteilt, die jeweils unterschiedlich gegenüber den Prozessoren angebunden sind. Da der Speicherzugriff der einzelnen Prozessorkerne somit unterschiedlich ausfallen kann, wird dies als Non-Uniform Memory Access-Architektur (NUMA-Architektur) bezeichnet. [7][5]

1.2.1. JUSUF

Für diese Arbeit wurde das HPC System JUSUF[4] (Jülich Support for Fenix), das insgesamt mit 209 Knoten ausgestattet ist, genutzt. Das System besitzt 4 Login-Knoten, 61 beschleunigte Knoten, die neben CPUs zusätzlich mit dem Grafikprozessor NVIDIA V100 ausgestattet sind und 144 Standard-Rechenknoten, die nur CPUs besitzen.

Jeder Knoten besitzt zwei Sockets mit jeweils einem AMD EPYC 7742 Prozessor. Der EPYC 7742 ist ein 64-Bit 64-Core-x86-Server-Mikroprozessor, der 2019 von AMD eingeführt wurde. Dieser Multi-Chip-Prozessor, der auf der Zen 2 Mikroarchitektur basiert, besitzt eine Basisfrequenz von 2,25 GHz und eine Boost-Frequenz von bis zu 3,4 GHz. Der Prozessor besitzt 64 physische Kerne, die zusätzlich jeweils über einen virtuellen Kern verfügen.

Der interne Aufbau dieser Generation von AMD Prozessoren hat eine komplexe hierarchische Struktur: Vier physische Kerne teilen sich einen L3-Cache von 16 MB und bilden einen sogenannten Core-Complex (CCX). Zwei dieser Core-Complexe bilden wiederum einen Core-Complex-Die (CCD). Zwei CCD bilden zusammen einen Quadranten mit einem eigenen Arbeitspeicher- und einer I/O-Anbindung (NUMA-Domains). Somit besitzt ein AMD EPYC 7742-Prozessor einen L3-Cache von insgesamt 256 MiB. An dieser Stelle ist festzuhalten, dass lediglich die vier Kerne innerhalb des CCX Zugriff auf den dazugehörigen L3-Cache haben und dieser nicht von Kernen ausserhalb dieser CCX Einheit verwendet werden kann. Die folgenden Abbildungen zeigen den Aufbau des Prozessors:

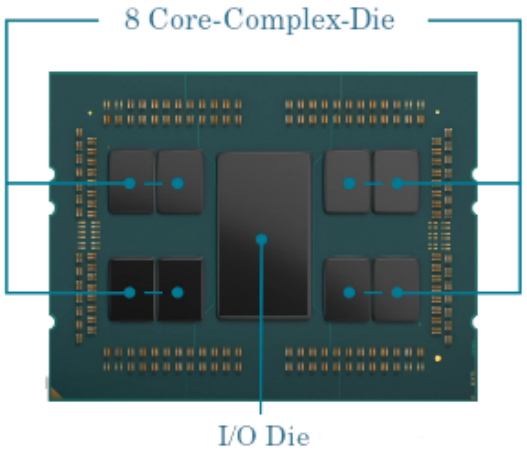


Abbildung 1.2.: Die-Aufbau des AMD EPYC Prozessors[1]

[4] https://apps.fz-juelich.de/jsc/hps/jusuf/index.html

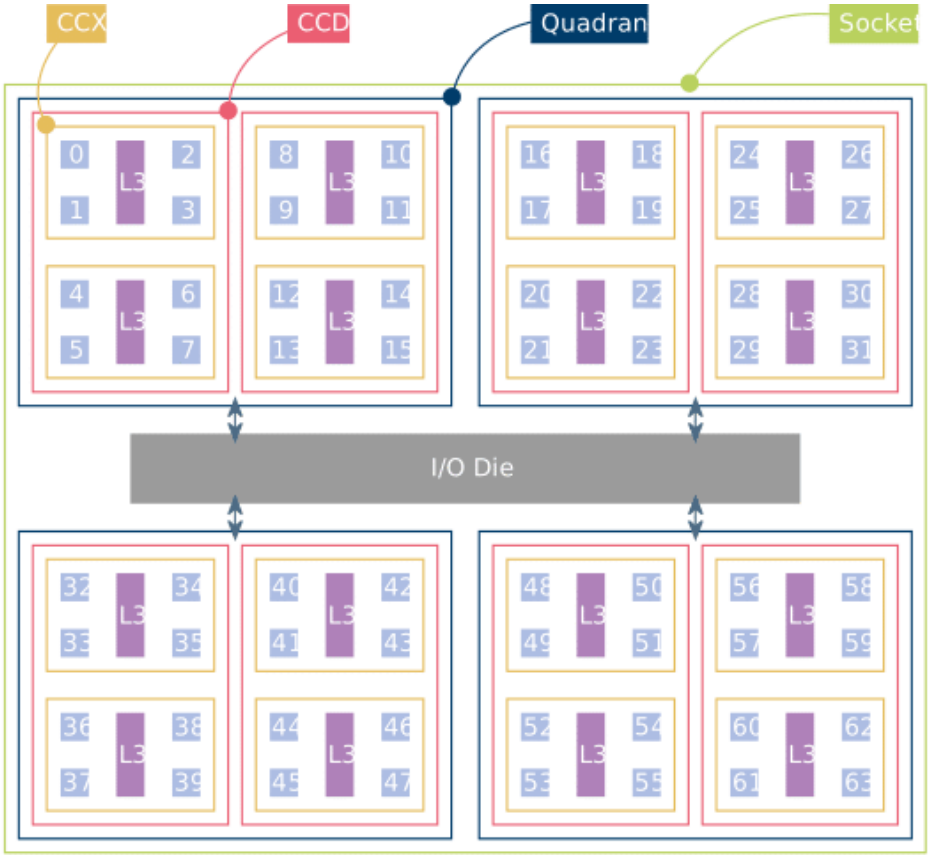


Abbildung 1.3.: Schematischer Aufbau des AMD EPYC 7742-Prozessors

Die folgende Grafik veranschaulicht den Aufbau und das Prozess-Pinning auf dem HPC System JUSUF.

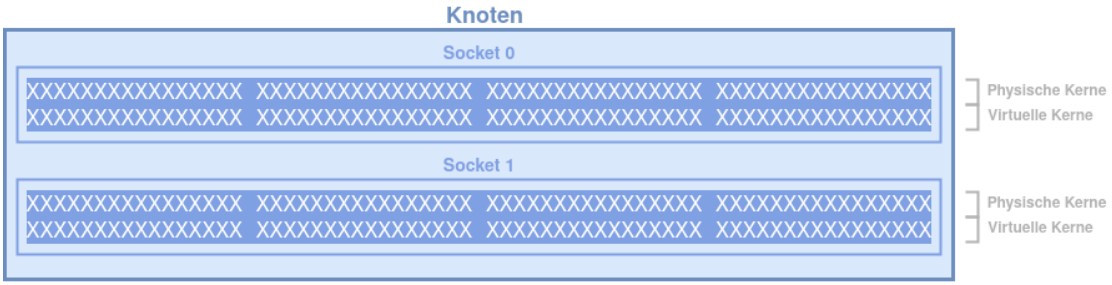


Abbildung 1.4.: Ein Rechenknoten auf JUSUF

Die 'X'-Zeichen repräsentieren die unbelegten Kerne des entsprechenden Sockets. Die obere Zeile stellt die physischen und die untere Zeile die virtuellen Kerne dar. Durch die Lücken zwischen den Kernen innerhalb eines Sockets sollen die einzelnen NUMA-Domains verdeutlicht werden. [10]

2. Hintergründe des Prozess Pinnings

2.1. Die Theorie des Prozess-Pinnings

Beim Ausführen eines Anwendungsprogramms kommt es nicht nur auf die Berechnung eines korrekten Ergebnisses an, sondern auch auf die Laufzeit, die die Anwendung benötigt um dieses Ergebnis zu berechnen. Bei einer Anwendung, die mit mehreren Prozessen oder Threads gestartet wird, ist eine parallele Ausführung durch Verteilung der Prozesse oder Threads auf verschiedene physische und virtuelle Kerne möglich. Prozess-Pinning, das Binden von Threads oder Prozessen an die Kerne eines Prozessors ist eine fortschrittliche Methode, um zu steuern, wie die Threads oder Prozesse innerhalb eines Systems verteilt werden. Dadurch können beispielsweise die Anzahl an lokalen Speicherzugriffen erhöht und somit kostspielige Remote-Speicherzugriffe vermieden werden, indem Prozesse oder Threads möglichst lokal zueinander gehalten werden. Dies kann beispielsweise die Leistung einer Anwendung verbessern. Der Begriff Affinitätsmaske beschreibt, die durch Konfigurationen veränderbare, ausgewählte Gruppe an Kernen, um die zugeordneten Threads eines Prozesses auszuführen. Diese Affinitätsmaske wird in den Beispielen dieser Arbeit wie folgt dargestellt:

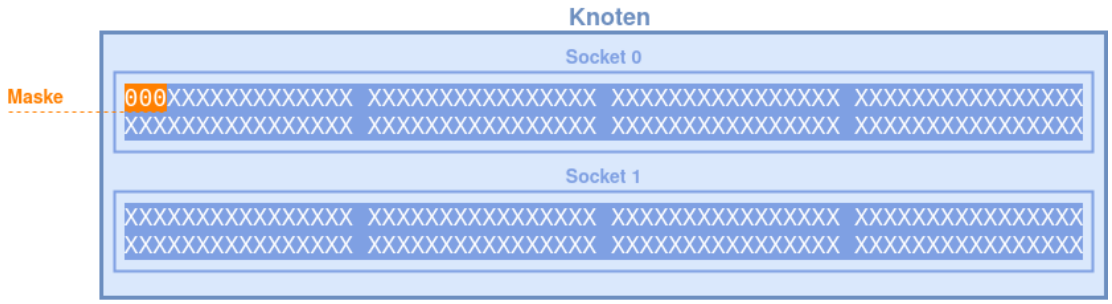


Abbildung 2.1.: Darstellung der Affinitätsmaske

Mithilfe der orange markierten Kerne sollen die belegten Kerne mit der entsprechenden Tasknummer dargestellt werden. In Abbildung 2.1 ist somit ein Task, der aus drei Threads besteht, zu sehen. Diese werden in diesem Beispiel auf die ersten drei physischen Kerne des ersten Sockets gebunden. Die genaue Reihenfolge der Threads innerhalb einer Task wird nicht weiter betrachtet. Die Konfiguration der Affinitätsmaske wird auf den HPC Systemen des JSC über eine angepasste Form des Workload Managers Slurm (Simple Linux Utility for Resource Management) geregelt. [13]

2.1.1. Non-Uniform Memory Access

Moderne Prozessoren können Daten oftmals deutlich schneller verarbeiten, als der an sie gebundene Arbeitsspeicher diese übertragen kann. In den traditionellen Symmetric Multiprocessing-Architekturen (SMP) greifen alle Prozessoren über einen Front Side Bus, auch bekannt als „North Bridge“, auf denselben Speicher zu. Wenn die Anzahl der Prozessoren wächst, gerät der Speicherbus in einen Systemengpass und die Prozessoren

beginnen zu „verhungern“, da diese darauf warten, dass Daten aus dem Speicher ankommen. Dies sorgt für verringerte Leistung und somit für langsamere Anwendungen.

Um dieses Problem zu lösen, wurde NUMA entwickelt, der den Speicher in mehrere Speicherknoten, sogenannte NUMA-Domains, unterteilt, die für einen oder mehrere Prozessoren lokal sind. Auf den lokalen Speicherknoten kann schneller zugegriffen werden als auf die anderen Speicherknoten.

Aus dieser Perspektive kann ein NUMA-System als eine Reihe von SMP-Systemen betrachtet werden: Jede NUMA-Domain fungiert als SMP-System. NUMA-Domains sind über eine Art Systemverbindung miteinander verbunden. Eine Kreuzschiene oder eine Punkt-zu-Punkt-Verbindung sind die gebräuchlichsten Arten solcher Verbindungen. Typisch ist diese NUMA-Architektur auch für die meisten x86-Server, welche oft auch in HPC Systemen genutzt werden und die oftmals über mehr als einen CPU-Sockel verfügen.

Bei einem NUMA-System wird die beste Leistung erzielt, wenn sich ein Prozess und sein Speicher in derselben NUMA-Domain befinden. [4]

2.1.2. GPU-Pinning

Die Konfiguration des Prozess-Pinnings wird normalerweise über die einzelnen Sockets beeinflusst bzw. werden die Prozesse und Threads im Fall eines NUMA-Systems, in Reihenfolge der NUMA-Domains an die Kerne gebunden. Auf den Systemen des JSC, die zusätzlich über Grafikprozessoren verfügen, ist die Reihenfolge der NUMA-Domains jedoch nicht aufsteigend sortiert, sondern anhand der Affinität zu den vorhandenen Grafikprozessoren (GPUs).

Die folgende Abbildung soll die Affinität der NUMA-Domains und der GPUs auf dem HPC-System JURECA veranschaulichen:

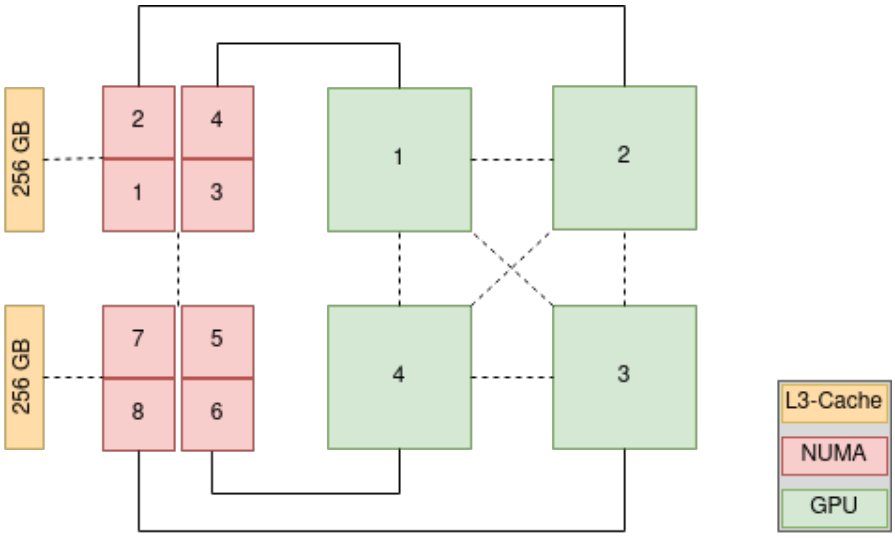


Abbildung 2.2.: Prozess-Pinning anhand der GPU auf JURECA

Die dritte NUMA-Domain hat Affinität mit der ersten GPU, sodass diese in der Reihenfolge am Anfang steht. Folgendermaßen sind die NUMA-Domains 2, 8 und 6 als nächstes an der Reihe. Da die restlichen NUMA-Domains keine direkte Affinität zu

den GPUs besitzen, bekommen diese die Rangfolge der NUMA-Domain, die ihnen am nächsten ist. Somit ist die schlussendliche Reihenfolge für alle NUMA-Domains `4, 2, 8, 6, 3, 1, 7, 5`. Dieses Pinning-Verhalten gilt derzeit für das HPC System JURECA, sowohl auf der Partition DC als auch GPU und das System JUWELS (Partiton Booster). Aus administrativen Gründen wurde dieses Verhalten für das JUSUF-System übernommen, da JUSUF den gleichen Prozessor und eine GPU an NUMA-Domain vier angebunden hat. Die folgende Grafik stellt das Pinning-Verhalten von acht Tasks und je einem Thread mit der Standard Bindungs- und Verteilungsoption dar, um die Reihenfolge der NUMA-Domains zu verdeutlichen:

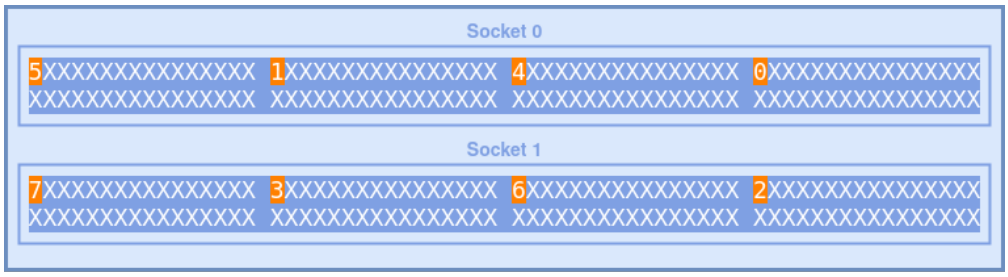


Abbildung 2.3.: Pinning-Verhalten auf JUSUF für die Reihenfolge der NUMA-Domains

2.2. Der Ressourcenmanager Slurm

Auf die HPC Systeme des JSC wird über einen dedizierten Satz von Login-Knoten zugegriffen, welche zum Kompilieren von Anwendungen sowie zur Vor- und Nachbearbeitung von Simulationsdaten verwendet werden. Der Zugriff auf die Rechenknoten wird von einem Workload-Manager gesteuert.

Auf den HPC Systemen kommt der, seitens der Firma Partec[1] angepasste Workload Manager PsSlurm zum Einsatz. Dies ist eine Variante basierend auf dem kostenlosen Open-Source-Ressourcenmanager und Batch-System Slurm[2]. Slurm ist ein modernes, erweiterbares Batch-System, das weltweit auf Clustern unterschiedlicher Größe weit verbreitet ist. Slurm kombiniert die Funktionalität des Batch-Systems und des Ressourcenmanagements.

Slurm besteht aus mehreren Diensten und Programmen. Der Dienst `slurmctld` ist für die Überwachung der verfügbaren Ressourcen und die Planung von Batch-Jobs verantwortlich. `slurmctld` läuft dabei mit einem speziellen Setup auf einem Verwaltungsknoten, um die Verfügbarkeit bei Hardware-Ausfällen zu gewährleisten. Benutzerprogramme wie `srun`, `sbatch`, `salloc` und `scontrol` interagieren mit dem Dienst `slurmctld`. In Slurm ist ein Job eine Zuweisung ausgewählter Ressourcen für einen bestimmten Zeitraum. Mit `sbatch` und `salloc` kann eine Jobzuordnung angefordert werden. Innerhalb eines Jobs können mit `srun` mehrere Jobschritte ausgeführt werden, die alle oder eine Teilmenge der zugewiesenen Rechenknoten verwenden. Jobschritte können gleichzeitig ausgeführt werden, wenn die Ressourcenzuordnung dies zulässt. Das folgende Beispiel zeigt einen Code-Ausschnit, in dem ein Jobschritt innerhalb eines Jobs

[1] https://par-tec.com/
[2] https://slurm.schedmd.com/

initiiert wird.

```
srun --nodes=1 --tasks-per-node=48 --cpus-per-task=1 ./executable
```

Listing 2.1: Initiieren eines Jobschrittes innerhalb eines Jobs

Der Befehl `srun` kann mit mehreren Argumenten ausgeführt werden und erhält den Pfad zur auszuführenden Datei (in diesem Beispiel `./executable`). Die benötigte Knotenanzahl wird durch das Argument `--nodes` (abgekürzt `-N`) festgelegt. Durch `--tasks-per-node` (abgekürzt `-n`) und `--cpus-per-task` (abgekürzt `-c`) wird festgelegt, wie viele Teiljobs (s.g. Tasks) auf einem Knoten laufen und wie viele Threads diese jeweils zugeordnet bekommen. Die Verteilung der Tasks und Threads auf die verfügbaren Kerne des jeweiligen Knotens, also die Prozessaffinität, kann durch die Optionen `--cpu-bind`, `--distribution` und `--hint` vom Benutzer beeinflusst werden. Die Option `--hint` legt fest, welche Kerne genutzt werden können. Der Standard (`--hint=-`) nutzt dabei alle verfügbaren Kerne eines Knotens, wobei die Option `nomultithread` nur die physischen Kerne nutzt. Die Bindungs- und Verteilungsoptionen werden in den folgenden Unterkapiteln grob beschrieben. Dabei wird hauptsächlich auf die genutzten Optionen dieser Arbeit eingegangen. Für eine ausführliche Beschreibung wird auf meine Seminararbeit[14] verwiesen. [3]

2.2.1. Bindungsoptionen

Mithilfe des Argumentes `--cpu-bind` steuert der Benutzter die Verteilung der Task auf die verfügbaren Kerne. [13]

2.2.1.1. Bindungsoption threads

Die aktuelle Standardoption des JSC ist `threads`, bei der jeder Task die angeforderte Threadanzahl erhält. Die Verteilung der einzelnen Threads auf die Kerne wird dabei mit Hilfe der Verteilungsoptionen (`--distribution`) gesteuert. Dieses und jedes nachfolgende Beispiel zeigt einen Knoten mit acht Tasks und jeweils drei Threads und den Standardverteilungsoptionen auf dem System JUSUF. Auf diesem System gilt die Reihenfolge `4, 2, 8, 6, 3, 1, 7, 5` für die NUMA-Domains.

Abbildung 2.4.: Bindungsoption `threads`

2.2.1.2. Bindungsoption cores

Bei der Bindungsoption `cores` werden die Tasks zu Anfang genauso zugewiesen wie in der Bindungsoption `threads`. Die endgültige Affinitätsmaske für einen Task umfasst

schlussendlich aber auch die virtuellen Kerne der belegten physischen Kerne. Somit werden die Tasks wie folgt auf die Kerne verteilt:

Abbildung 2.5.: Bindungsoption `cores`

Bei dieser Bindungsoption kann es zu Mehrfachbelegung der Kerne kommen, wobei nicht weiter festgelegt wird, welcher Thread welchen Kern nutzt. Dies übernimmt das Betriebssystem dynamisch oder kann durch weitere Einstellungsmöglichkeiten des Nutzers vorgegeben werden.

2.2.1.3. Bindungsoption rank

Zudem existiert die Bindungsoption `rank`, die nicht von der Verteilungsoption beeinflusst wird. Bei dieser Option werden die Kerne auch socketübergreifend, nacheinander mit den Tasks, unter Beachtung der NUMA-Domain Reihenfolge belegt. Dabei werden zuerst alle physischen und danach die virtuellen Kerne belegt.

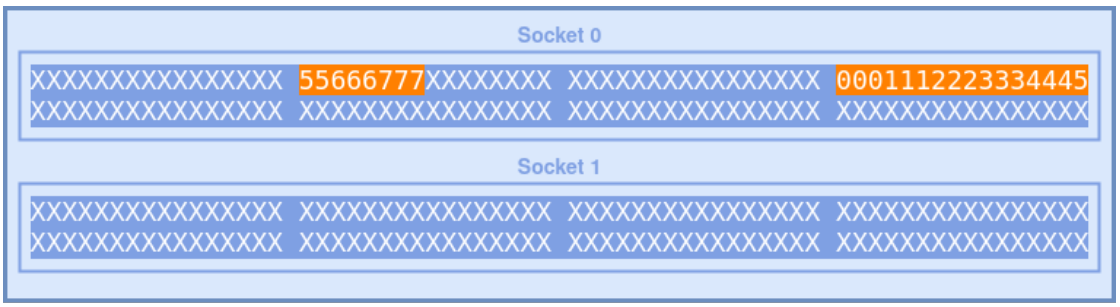


Abbildung 2.6.: Bindungsoption `rank`

2.2.1.4. Bindungsoption rank_ldom

Zuletzt gibt es die Bindungsoption `rank_ldom`, die genauso wie `rank` nicht von den Verteilungsoptionen beeinflusst wird. Das `ldom` der Bindungsoption steht für 'local domains' und beschreibt die Verteilung der Task. Dabei belegen alle Threads einer Task, solange es möglich ist, nur Kerne innerhalb eines Speicherknotens. Die Abbildung zeigt die Bindung der Task an die Kerne:

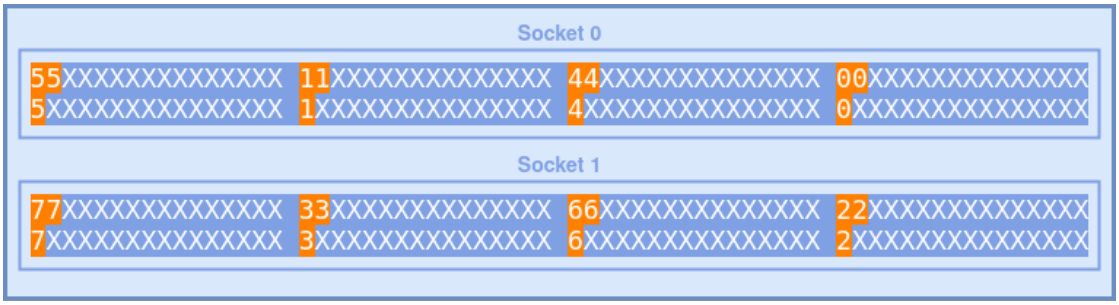


Abbildung 2.7.: Bindungsoption `rank_ldom`

2.2.2. Verteilungsoptionen

Der Benutzter kann mithilfe des Arguments `--distribution` die Verteilung der Tasks bzw. Threads beeinflussen. Dabei kann die übergebende Zeichenfolge aus insgesamt vier Teilen bestehen, die durch Doppelpunkt und Komma getrennt werden. Folgendes Beispiel zeigt die allgemeine Form des Argumentes:

```
1 --distribution=<knoten_ebene>:<socket_ebene>:<kern_ebene>,<pack>
```

Listing 2.2: Allgemeine Form der Verteilungsoption `--distribution`

Der erste und vierte Teil der Zeichenkette beeinflusst die Verteilung der Threads auf den Knoten. Da sich diese Arbeit nur mit dem Pinning für einen Knoten befasst, werden diese Teile nicht weiter erläutert. Die für diese Arbeit relevanten Teile der Zeichenfolgende werden im Folgenden beschrieben. [13]

2.2.2.1. Socket-Ebene

Der zweite Teil der `--distribution`-Option beschreibt die Aufteilung der Task auf die Sockets bzw. Speicherknoten. Es gibt drei Auswahlmöglichkeiten für diese Option: `cyclic`, `block` und `fcyclic`. Diese wird im Folgenden grob beschrieben und die Verteilung durch Grafiken auf dem System JUSUF verdeutlicht.

Die Option `cyclic` ist die Standardoption (die Standardoptionen der verschiedenen Ebenen der Verteilungsoptionen können jeweils mit `*` abgekürzt angegeben werden) am JSC. Die Verteilung der Threads steuert der dritte Teil der `--distribution`-Option. Die folgende Grafik zeigt insgesamt neun Tasks und drei Threads mit der `--distribution`-Option `cyclic` auf Socket-Ebene.

Abbildung 2.8.: Verteilungsoption `cyclic` auf Socket-Ebene (`--distribution=*:cyclic:*`)

Es ist zu sehen, dass die nächste Task erneut dem ersten Speicherknoten zugeordnet wird, sobald alle Speicherknoten einmal belegt wurden.

`block` ist ebenfalls eine Option, bei der die Task einem Socket bzw. Speicherknoten zugeordnet wird, bis dieser 'gefüllt' ist. Erst dann wird der nächste Socket bzw. Speicherknoten verwendet. Die folgende Grafik soll die Verteilung verdeutlichen und zeigt insgesamt neun Tasks mit jeweils fünf Threads mit der `--distribution`-Option `block` auf Socket-Ebene:

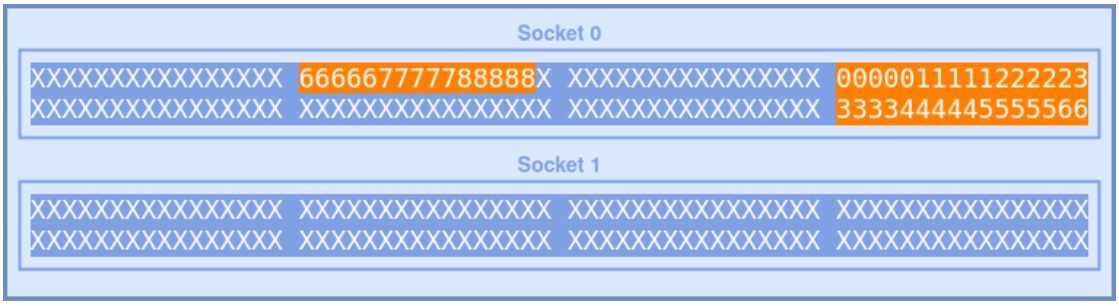


Abbildung 2.9.: Verteilungsoption `block` auf Socket-Ebene (`--distribution=*:block:*`)

Zuletzt wird die Option `fcyclic` beschrieben, welche die Threads-Verteilung steuert. Jeder Thread einer Task wird dabei dem jeweils nächsten Socket bzw. Speicherknoten zugeordnet. Abbildung 2.10 zeigt die Verteilung der Option `fcyclic` auf Socket-Ebene mit acht Tasks und jeweils drei Threads pro Task:

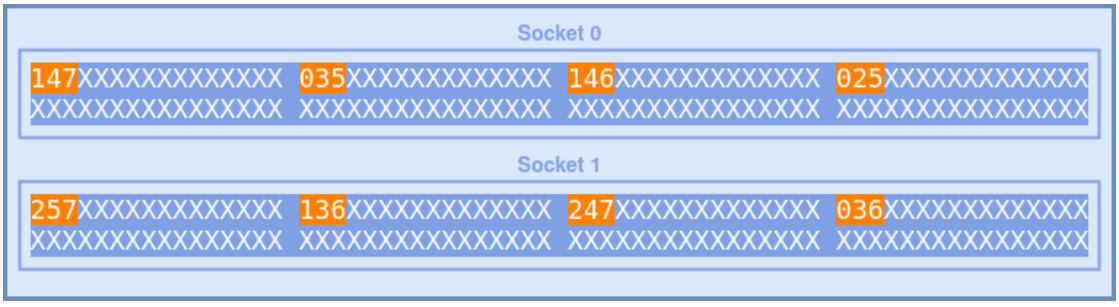


Abbildung 2.10.: Verteilungsoption `fcyclic` auf Socket-Ebene (`--distribution=*:fcyclic:*`)

2.2.2.2. Kern-Ebene

Die dritte Option innerhalb von `--distribution` beschreibt die Verteilung der Threads auf den Kernen der vorher bestimmten Sockets. Auch bei dieser Option gibt es drei Verteilungsoptionen, die im Folgenden beschrieben und durch Abbildungen des schematischen Aufbaus der in JUSUF verbauten Rechenknoten und deren CPUs verdeutlicht werden.

Die Standardoption am JSC ist `fcyclic`, bei der die Threads der Reihe nach den physischen Kernen zugeordnet werden. Erst wenn alle physischen Kerne belegt sind, werden die Threads den virtuellen Kernen, ebenfalls der Reihe nach, zugeordnet. Ab-

bildung 2.11 zeigt die Verteilung von neun Tasks und jeweils drei Threads mit der Standardoption `fcyclic`:

Abbildung 2.11.: Verteilungsoption `fcyclic` auf Kern-Ebene (`--distribution=*:*:fcyclic`)

Bei der Option `cyclic` werden die Threads einer Task dem ganzen Kern, also sowohl dem physischen als auch dem virtuellen Kern der Reihe nach zugeordnet, was die folgende Abbildung mit neun Tasks und drei Threads pro Task veranschaulicht:

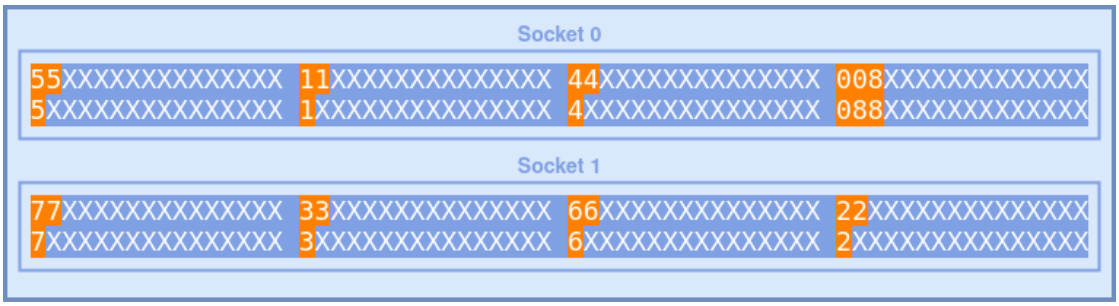


Abbildung 2.12.: Verteilungsoption `block` auf Kern-Ebene (`--distribution=*:*:block`)

Die folgende Abbildung zeigt die Option `cyclic` auf Kern-Ebene:

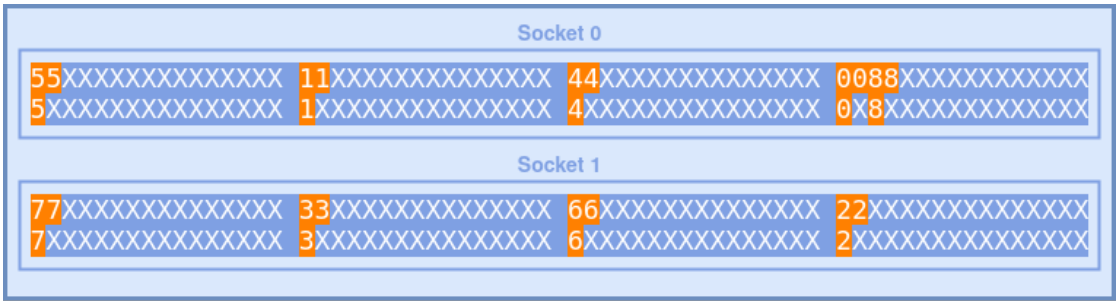


Abbildung 2.13.: Verteilungsoption `cyclic` auf Kern-Ebene (`--distribution=*:*:cyclic`)

Anhand der Task acht ist zu sehen, dass die Threads zunächst wie bei der Option `block` zugeordnet werden, jedoch bei einer neuen Task auf dem physischen Kern beginnen.

2.2.3. Verifizierung des Pinning-Verhaltens

Die Erklärungen des Pinning-Verhaltens wurde der Dokumentation des HPC Systems JUSUF[13] entnommen, welches die einzelnen Bindungs- und Verteilungsoptionen des Ressourcenmanagers Slurm anhand von Abbildungen und Erklärungen aufzeigt. Da Slurm auf allen HPC Systemen des JSC angewendet wird, ist davon auszugehen, dass die Bindungen und Verteilungen für alle Systeme gelten.

Um dieses Pinning-Verhalten für das HPC System JUSUF zu verifizieren, habe ich bereits während meiner Seminararbeit zur Entwicklung einer webbasierten Visualisierung des Pinning-Verhaltens auf verschiedenen HPC Systemen eine Option des Befehls `srun` genutzt. Dieser gibt eine Hexadezimale-Maske aus, mithilfe derer das tatsächliche Pinning der einzelnen Threads erkennbar wird. Dafür muss zusätzlich zur Bindungsoption `--cpu-bind` das Argument `verbose` mitgegeben werden. Ein `srun`-Befehl könnte dann wie folgt aussehen:

```
1 srun -n 4 -c 1 --cpu-bind=threads,verbose --distribution=*:*:* hostname
    |& sort
```

Listing 2.3: `srun`-Befehl zur Ermittlung der Hexadezimal-Pinning-Maske

Dieser `srun`-Befehl führt vier Tasks und jeweils einem Thread auf standardmäßig einem Knoten aus. Dafür benutzt dieser die Bindungsoption `threads` und die Standardverteilungsoption. Um eine Ausgabe zu erzeugen, soll der Hostname sortiert ausgegeben werden. Dieser Befehl sorgt für folgende Ausgabe:

```
1 cpu_bind=THREADS - jsfc098, task  0  0 [12219]: mask 0x1000000000000 set
2 cpu_bind=THREADS - jsfc098, task  1  1 [12220]: mask 0x10000 set
3 cpu_bind=THREADS - jsfc098, task  2  2 [12221]: mask 0
    x10000000000000000000000000000000 set
4 cpu_bind=THREADS - jsfc098, task  3  3 [12222]: mask 0
    x10000000000000000000 set
5 jsfc098
6 jsfc098
7 jsfc098
8 jsfc098
```

Listing 2.4: Ausgabe des `srun`-Befehl zur Ermittlung der Hexadezimal-Pinning-Maske

Zeilen fünf bis acht sind nicht relevant, da diese lediglich den Hostname ausgeben, jedoch geben die ersten vier Zeilen die relevanten Informationen über das Pinning wieder. Einerseits wird die Bindungsoption und der Knoten, auf dem die Task ausgeführt wird, ausgegeben. Andererseits wird die Tasknummer und die dazugehörige Maske ausgegeben. Mit Hilfe eines Skripts oder einer eingebauten Funktion auf dem Pinning-Webtool[3] kann die Hexadezimale-Maske in eine binäre Darstellung umgewandelt werden, sodass die in der Hexadezimale-Maske codierte Belegung der Kerne einfacher zu erkennen ist.

[3] https://apps.fz-juelich.de/jsc/llview/pinning/

3. Verwendete Softwarepakete

3.1. Die Benchmarking Umgebung JUBE

Das Benchmarking einer Anwendung umfasst normalerweise zahlreiche Aufgaben, die mehrere Durchläufe verschiedener Konfigurationen beinhalten. Das Konfigurieren, Kompilieren und Ausführen einer Anwendung auf mehreren Plattformen sowie die Ergebnisverifizierung und -analyse erfordert viel administrativen Aufwand und erzeugt viele Daten, die analysiert und gesammelt werden müssen. Ohne eine Benchmarking Umgebung müssen diese Schritte von Hand durchgeführt werden.

Durch die Automatisierung von Benchmarks ist es möglich, diese zu reproduzieren und zu vergleichen, was bei der Durchführung im Vordergrund steht. Darüber hinaus ist die Verwaltung unterschiedlicher Parameterkombinationen fehleranfällig und verursacht oft einen erheblichen Arbeitsaufwand, insbesondere wenn der Parameterraum groß wird.

Die Programmumgebung JUBE[11] hat die Möglichkeit, Benchmarks systematisch durchzuführen und zu analysieren. JUBE wurde vom JSC des Forschungszentrum Jülich entwickelt und wird immer noch aktiv weiterentwickelt. Es ermöglicht die Anpassung benutzerdefinierter Arbeitsabläufe an neue Architekturen. Somit können Fehler und Arbeitsaufwand bei verschiedenen Kombinationen vermieden werden.

Für jede Anwendung werden die Benchmark-Daten in einem bestimmten Format ausgeschrieben, sodass es JUBE möglich ist, die gewünschten Informationen abzuleiten. Diese Daten können durch automatische Vor- und Nachbearbeitungsskripte extrahiert und visualisiert werden.

Die Benchmarking Umgebung JUBE bietet ein skriptbasiertes Framework, um Benchmark-Sets zu erstellen. Diese Sets können auf verschiedenen Computersystemen ausgeführt und die Ergebnisse ausgewertet werden.

In dieser Arbeit wurde JUBE zur Automatisierung der Benchmarks, die zum Leistungsvergleich benötigt werden, genutzt. Dabei automatisiert JUBE den Arbeitsablauf aus Kompilieren, Konfigurieren und Ausführen der einzelnen Benchmark-Läufe. In den folgenden Unterkapiteln werden die genutzten JUBE-Optionen zum Automatisieren der Benchmarks beschrieben.[9][8]

3.1.1. Eingabedatei

JUBE unterstützt zwei verschiedene Arten von Eingabeformaten: XML- und YAML-basierte Dateien. Beide Formate unterstützen die gleiche Menge an JUBE-Funktionen. Für die Eingabedateien dieser Arbeit wurde das XML-basierte Eingabeformat genutzt. Die allgemeine Struktur der Eingabedatei sieht wie folgt aus:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <jube>
3   <benchmark name="name_des_benchmarks" outpath="ausgabe_pfad">
4     <comment>Beispielkommentar</comment>
5   </benchmark>
6 </jube>
```

Listing 3.1: Allgemeine Struktur einer JUBE-Eingabedatei im XML-Format

Jede XML-basierte JUBE-Eingabedatei beginnt (nach der allgemeinen XML-Kopfzeile) mit dem Root-Tag `<jube>`. Das erste Tag, welches Benchmark-spezifische Informationen enthält, ist `<benchmark>`. Das Attribut `outpath` beschreibt das Benchmark-Laufverzeichnis (relativ zur Position der Eingabedatei) und `name` deklariert den Namen des Benchmarks. Mit dem Tag `<comment>` können Benchmark-bezogene Kommentare im Benchmark-Verzeichnis gespeichert werden.

3.1.2. Parameterräume

JUBE besitzt die Möglichkeit, eigene Parameterräume zu generieren. Dafür werden Parametersets mit dazugehörigen Parametern nach folgender Struktur erstellt:

```
<parameterset name="parameterset_name">
  <parameter name="parameter_name" type="int">1,2,4</parameter>
</parameterset>
```

Listing 3.2: Beispiel für ein Parameterset in JUBE

Der Name des Parametersets wird durch das Attribut `name` festgelegt und kann darüber später an verschiedenen Stellen (über `$parameter_name`) benutzt werden. Durch das Attribut `type` kann der Typ des Parameters festgelegt werden. In diesem Beispiel ist es der Typ `int`, also eine Ganzzahl. Durch die Abtrennung der Werte mittels eines Kommas (dem sogenannten `separator` , der durch ein Attribut verändert werden kann) wird dem Parameter mitgeteilt, dass es sich um mehrere mögliche Werte handelt. Somit wird ein Befehl, der diesen Parameter benutzt, mehrfach ausgeführt, um alle Parameterkombinationen zu nutzen. In diesem Fall würde der Befehl dreimal mit dem Wert eins, zwei oder vier durchlaufen. Zusätzlich gibt es noch das Attribut `mode`, mit dem die Skriptsprache des Parameters festgelegen werden kann und somit bspw. Python-Befehle benutzt werden können. Dieses Feature wurde im Wesentlichen für das Konfigurieren der Task- und Threadanzahl genutzt.

3.1.3. Ausführungsschritte

Mittels eines Ausführungsschrittes können Shell-Kommandos durch JUBE ausgeführt werden. Dabei können Parametersets mithilfe des `<use>`-Tags innerhalb des Ausführungsschrittes benutzt werden und mit `<do>` die Kommandos, die ausgeführt werden sollen, definiert werden. Die Struktur eines Ausführungsschrittes sieht wie folgt aus:

```
<step name="step_name">
  <use>parameterset_name</use>
  <do>echo $parameter_name</do>
</step>
```

Listing 3.3: Beispiel für einen Ausführungsschritt in JUBE

Abhängigkeiten können zwischen mehreren Ausführungsschritten mit Hilfe des Attributes `depend` angegeben werden, sodass ein Ausführungsschritt erst nach Abschluss des im `depend`-Attribut genannten Ausführungsschrittes gestartet wird. Die Ausführungsschritte in JUBE wurden für das Kompilieren und Ausführen der Anwendungen genutzt.

3.1.4. Dateien und Substitutionen

Jeder Schritt wird in einem eindeutigen, automatisch von JUBE verwalteten Verzeichnis ausgeführt. Normalerweise werden in diesem Verzeichnis externe Dateien (z. B. die Quelldateien) benötigt und bestimmte Parameter innerhalb einer Datei sollen substituiert werden. Es gibt zwei zusätzliche Set-Typen, die dieses Verhalten innerhalb von JUBE handhaben:

Durch ein Fileset können Dateien verlinkt oder kopiert werden:

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

Listing 3.4: Beispiel für ein Fileset in JUBE

In diesem Beispiel wird die Datei `datei.in` aus dem aktuellen Arbeitsverzeichnis in das Verzeichnis des Schrittes kopiert, sodass dieses dort genutzt werden kann. Desweiteren gibt es ein Substitutionsset, welches den Substitutionsprozess einer Datei beschreibt:

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

Listing 3.5: Beispiel für ein Substituteset in JUBE

Das `<iofile>` enthält den Namen der Eingabe- und Ausgabedatei. Der Pfad ist relativ zum Verzeichnis des Schrittes. Der `<sub>`-Tag gibt die Ersetzung an. Alle Vorkommen von `source` werden durch `dest` ersetzt.

3.1.5. Statistische Patternräume

Zusätzlich zu Parametersets können auch Patternsets generiert werden, die bspw. zum Auslesen von Ergebnisgrößen genutzt werden können. Dabei wird ein Patternset mit den dazugehörigen Pattern ähnlich zu der Erstellung eines Paramtersets, wie folgt definiert:

```
<patternset name="pattern">
  <pattern name="number_pat" type="int">jube_pat_int</pattern>
</patternset>
```

Listing 3.6: Beispiel für ein Patternset in JUBE

Auch bei den Pattern kann der Name, Typ und Modus über die dazugehörigen Attribute definiert werden. Innerhalb des Patterntags kann mithilfe von regulären Ausdrücken oder internen, vordefinierten JUBE-Variablen (in diesem Beispiel `$jube_pat_int` für einen ganzzahligen Wert) das zu suchende Pattern deklariert werden. Bei dem mehrfachen Auffinden des genutzten Patterns erstellt JUBE automatisch statistische Variablen wie beispielsweise den Durchschnitt, das Minimum oder Maximum der gefundenen Pattern.

3.1.6. Analyse und Ergebnisse

Das Ausführen der Benchmarks erzeugt mehrere Verzeichnisse. JUBE bietet die Möglichkeit, die über diese Verzeichnisse verteilten Ergebnisdateien zu durchsuchen, um

relevante Daten zu extrahieren und eine Ergebnistabelle zu erstellen. Mit der Hilfe des `<analyser>`-Tags, der die angegebenen Pattern verwendet, können beliebige Ergebnisse (bspw. aus der Standardausgabe) extrahiert werden:

```
<analyser name="analyse">
  <use>pattern</use>
  <analyse step="step_name">
    <file>stdout</file>
  </analyse>
</analyser>
```

Listing 3.7: Beispiel für eine Analyse in JUBE

Durch die Tags `<use>` und `<file>` kann das Patternset und die gewünschte Ausgabe deklariert werden. Daraufhin wird die angegebene Datei nach dem Pattern durchsucht.

Der letzte Schritt ist die Erstellung der Ergebnistabelle:

```
<result>
  <use>analyse</use>
  <table name="result" sort="parameter_name">
    <column>parameter_name</column>
  </table>
</result>
```

Listing 3.8: Beispiel für ein Ergebnistabelle in JUBE

Hier muss ein vorherige Analyse verwendet werden. `<column>` definiert eine Spalte der Ergebnistabelle und können Pattern- oder Parameternamen enthalten. Mithilfe von Attributen wie `sort` oder `style` kann die Darstellung der Ergebnistabelle verändert werden.

3.1.7. Externe Daten einbinden

Für eine bessere Struktur innerhalb einer JUBE-Eingabedatei ist es möglich schon vorhandene Sets wiederzuverwenden. Zum Einbinden gibt es verschiedene Möglichkeiten, wie in dem folgenden Codebeispiel zu sehen ist:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="einbinden">
    <parameterset name="param_set" init_with="externe_datei.xml">
      <parameter name="foo">bar</parameter>
    </parameterset>

    <step name="say_hello">
      <use>param_set</use>
      <use from="externe_datei.xml">param_set2</use>
      <do>echo $foo</do>
    </step>
  </benchmark>
</jube>
```

Listing 3.9: Beispiel für die Einbindung von externen Daten in JUBE

Anhand des Attributes `init_with` kann ein Set aus einer externen JUBE-Datei initialisiert werden. Zusätzlich können weitere Werte hinzugefügt oder schon vorhandene überschrieben werden.

Um ein Set aus einer externen JUBE-Datei innerhalb eines Ausführungsschritt vollständig und unverändert zu benutzten, kann der Tag `<use from=..>` genutzt werden.

3.2. Verwendete Benchmarks

Um die Leistung verschiedener Prozess-Pinning Strategien zu vergleichen, wurden zwei existierende Benchmarks ausgewählt und auf dem HPC System JUSUF ausgeführt. Im Folgenden werden die genutzten Benchmarks und deren Ergebnisgrößen beschrieben.

3.2.1. STREAM-Benchmark

Der STREAM-Benchmark[1] ist ein einfaches synthetisches Benchmark-Programm, welches die Speicherbandbreite und die benötigte Laufzeit für einfache Vektoroperationen misst.

Der STREAM-Benchmark wurde speziell für die Arbeit mit Datensätzen entwickelt, die viel größer sind als der verfügbare Cache. Somit steht hier vor allem die Arbeitsspeicheranbindung im Vordergrund, jedoch kann auch die Analyse einzelner Cache-Level möglich sein.

3.2.1.1. Algorithmus und Ergebnisgrößen

Der STREAM-Benchmark ist ein in Fortran77 bzw. C geschriebenes Programm, der die Leistung von vier Vektoroperationen misst. Für den parallelen Betrieb stehen MPI zum Einsatz mehrerer Tasks und OpenMP zum Einsatz mehrerer Threads zur Verfügung.

Der erste Operator ist der `COPY`-Operator. Dieser kopiert einen Vektor und misst die Übertragungsraten ohne Arithmetik ($\vec{a} = \vec{b}$). Der `SCALE`-Operator wendet vor dem Kopieren eine arithmetische Operation (Multiplikation mit einer Konstante) an ($\vec{a} = \vec{b}$*k). Der dritte Operator `SUM` summiert zwei Vektoren, um mehrere Lade- bzw. Speicherzugriffe zu testen ($\vec{a} = \vec{b} + \vec{c}$). Der letzte Operator `TRIAD` benutzt alle arithmetische Operationen der anderen Operatoren. Somit addiert er dem zu kopierenden Vektor einen skalierten weiteren Vektor zu und misst die Übertragungsrate ($\vec{a} = \vec{b} + \vec{c}$*k). Jeder Operator wird standardmäßig zehnfach durchgeführt, um eine bessere statistische Aussage über die Übertragungsrate geben zu können.

Die Vektor-Größen müssen dabei so festgelegt werden, dass jeder Vektor größer als der Cache des zu testenden Systems ist. Die Gleitkommaarithmetik selbst besitzt dabei nur einen geringen Einfluss auf die Messung, sodass die vier STREAM-Bandbreitenwerte relativ nah beieinanderliegen und sich vornehmlich an der Anzahl der nötigen Speicherzugriffe orientieren. Diese sind vor allem bei den Operatoren `SUM` und `TRIAD` größer, da hier zwei Vektoren zum Einsatz kommen.

Die Größe der Vektoren kann beliebig angepasst werden, um die nötigen Speicherzugriffe und damit den STREAM-Benchmark für ein bestimmtes HPC System zu optimieren.

Neben der Vektorgröße, der Anzahl der genutzten Tasks und deren Threadanzahl sieht der relevante Teil der Ausgabe des STREAM-Benchmarks wie folgt aus:

[1] https://www.cs.virginia.edu/stream/

```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           20077.3     1.850217     1.593838     2.258304
Scale:          20024.2     1.681914     1.598064     2.272105
Add:            23132.3     2.278303     2.075022     2.688537
Triad:          22709.0     2.246446     2.113698     2.467156
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Listing 3.10: Beispiel für die Ausgabe des STREAM-Benchmarks

Es werden die beste gemessene Speicherbandbreite und die Zeiten zum Durchlaufen der jeweiligen Messungen herausgeschrieben. Dabei wird bei den Zeiten das arithmetische Mittel der zehn Durchläufe, die minimalste und die maximalste Zeit ermittelt und ausgegeben. Außerdem wird die Fehlergröße der jeweiligen durchgeführten Operationen bemessen und ausgegeben. [12]

3.2.2. HPL-Benchmark

HPL[2] ist ein Softwarepaket, welches ein zufälliges, lineares Gleichungssystem in doppelter Genauigkeit (64 Bit) auf Rechnern mit verteiltem Speicher löst. Es kann somit als frei verfügbare Implementierung des Linpack Benchmark angesehen werden.

Das HPL-Paket bietet ein Test- und Messprogramm zur Quantifizierung der Genauigkeit der erhaltenen Lösung sowie der für ihre Berechnung benötigten Zeit. Die beste Leistung, die mit dieser Software auf einem System erzielt werden kann, hängt von einer Vielzahl von Faktoren ab. Nichtsdestotrotz ist der Algorithmus bezüglich seiner parallelen Effizienz sehr gut skalierbar, da die Speichernutzung pro Prozessor konstant gehalten werden kann. Aus diesem Grund kommt der HPL auch im Rahmen der TOP500 Liste[3] zur Bewertung der schnellsten Rechensysteme der Welt zum Einsatz.

Das HPL-Softwarepaket nutzt zur parallelen Berechnung MPI. Außerdem ist die Installation des Basic Linear Algebra Subprograms (BLAS[4]) nötig, welches wiederum OpenMP zur Parallelisierung über Threads nutzen kann.

3.2.2.1. Algorithmus

Der HPL-Benchmark erzeugt ein lineares System $A\vec{x} = \vec{b}$ der Ordnung n mit einer zufälligen quadratischen Matrix A und einem Vektor $\vec{b}$, der so gewählt ist, dass die Lösung des Systems der Vektor $\vec{x}$ ist, dessen Komponenten alle Einsen sind. Dadurch entsteht eine n-mal-n+1-Koeffizientenmatrix, die mithilfe der LU-Faktorisierung gelöst wird.

Die Daten werden nach einem blockzyklischen Schema auf ein zweidimensionales P-mal-Q-Raster von Prozessen verteilt, um einen Lastausgleich sowie die Skalierbarkeit des Algorithmus zu gewährleisten. Die n-mal-n+1-Koeffizientenmatrix wird zuerst logisch in nb-mal-nb-Blöcke unterteilt, die zyklisch auf das P-mal-Q-Prozessgitter verteilt werden. Dies geschieht in beiden Dimensionen der Matrix, wie die Abbildung 3.1 veranschaulicht wird.

[2] https://www.netlib.org/benchmark/hpl/
[3] https://www.top500.org/
[4] http://www.netlib.org/blas/

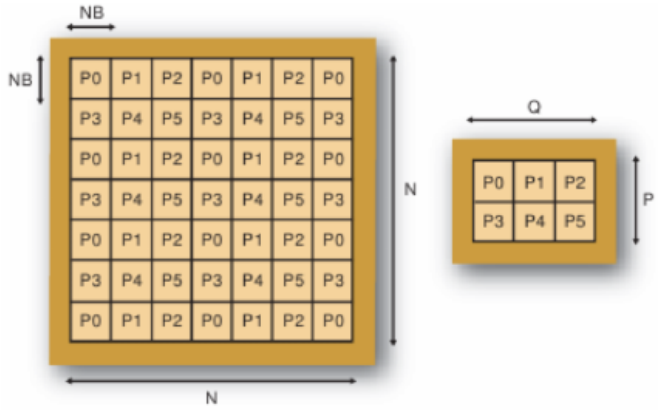


Abbildung 3.1.: Datenverteilung des HPL-Algorithmus [2]

Durch eine Eingabedatei können beispielsweise die Problemgrößen wie n, nb und P und Q an das jeweilige HPC System angepasst werden.

3.2.2.2. Ergebnisse

Durch die Eingabedatei lässt sich zudem der Ausgabeort der Ergebnisse anpassen, der standardmäßig die Konsolenausgabe ist. Die Ausgabe enthält alle Fehler- und Problemgrößen, die zuvor in der Eingabedatei festgelegt wurden. Zusätzlich werden die Ergebnisgrößen des jeweiligen Laufes ausgegeben. Im Folgenden ist ein Ausschnitt der Standardausgabe zu sehen, der die erzielten Ergebnisse eines Beispiellaufes zeigt:

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR01C2R4       32768    64     1     1             502.94           4.6641e+01
HPL_pdgesv() start time Fri Jun 25 18:16:39 2021

HPL_pdgesv() end time   Fri Jun 25 18:25:02 2021

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=2.29130495e-03PASSED
================================================================================

Finished      1 tests with the following results:
1 tests completed and passed residual checks,
0 tests completed and failed residual checks,
0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
================================================================================
```

Listing 3.11: Beispiel für die Ausgabe des HPL-Benchmarks

Wie zu sehen ist, wird sowohl die gemessene Laufzeit als auch die erreichten Gflops des jeweiligen Laufes ausgegeben. Zusätzlich wird der relative Fehler der Lösung berechnet und ausgegeben. [6]

4. Praktische Durchführung der Benchmarks

Um die Benchmarks auf dem HPC System JUSUF zu nutzen, wurde jeweils eine passende JUBE-Datei erstellt, die den Benchmark gegebenenfalls kompiliert und anschließend mit verschiedenen Konfigurationen auszuführt. In den folgenden Unterkapiteln wird dieses Vorgehen für die beiden Benchmarks im Detail erläutert.

4.1. Durchführung des STREAM-Benchmarks

Die University of Virginia bietet auf ihrer Internetseite[1] die Dokumentation und den Quellcode des STREAM-Benchmarks an. Es existiert bereits eine erweiterte Version, die STREAM2-Version, jedoch konzentriert sich diese Arbeit auf den ursprünglichen STREAM-Benchmark. Es gibt verschiedene Versionen des Quellcodes, wie beispielsweise eine MPI- oder OpenMP-Version. Da sowohl die Task- als auch die Threadanzahl konfiguriert werden soll, wurde für diese Arbeit die MPI-Version genutzt, da diese ebenfalls OpenMP Implementationen enthält. Der Quellcode blieb bei der Benutzung bis auf die Vektorgröße, unverändert. Die Vektorgröße, die standardmäßig über 10.000.000 Elemente verfügt, wurde auf 2.000.000.000 Elemente erhöht, um den Cache der Prozessoren auszulasten.

4.1.1. JUBE-Implementierung

Um das Kompilieren und Ausführen des STREAM-Benchmarks zu automatisieren, wurden verschiedene JUBE-Eingabedateien im XML-Format erstellt. Insgesamt wurden sieben verschieden Verteilungs- bzw. Bindungsoption getestet, die in der folgenden Tabelle aufgelistet sind:

Nr.	Bindungsoption	Verteilungsoption
1	`threads`	`distribution=block:cyclic:fcyclic` (entspricht `distribution=*:*:*`)
2	`cores`	`distribution=block:cyclic:fcyclic` (entspricht `distribution=*:*:*`)
3	`rank`	-
4	`rank_ldom`	(entspricht Bindungsoption `threads` und Verteilungsoption `distribution=*:*:block`)
5	`threads`	`distribution=block:block:fcyclic` (entspricht `distribution=*:block:*`)
6	`threads`	`distribution=block:cyclic:cyclic` (entspricht `distribution=*:*:cyclic`)
7	`threads`	`distribution=block:fcyclic:fcyclic` (entspricht `distribution=*:fcyclic:*`)

Damit jede Verteilungs- bzw. Bindungsoption ihr eigenes Verzeichnis hat, um die Daten leichter auslesen zu können, wurde für jede Option eine eigene JUBE-Eingabedatei

[1] http://www.cs.virginia.edu/stream/

erstellt. Diese enthält die Konfiguration der jeweiligen Verteilungs- und Bindungsoption. Im Folgenden werden anhand der `threads`-Bindungsoption mit der Standardverteilung die Konfigurationen und Ausführungen der JUBE-Eingabedatei beschrieben. Neben der allgemeinen Struktur enthält die Eingabedatei ein Parameterset, das wie folgt aufgebaut ist:

```
<parameterset name="systemParameter" init_with="platform.xml">
  <parameter name="nodes" type="int">1</parameter>
  <parameter name="taskspernode" type="int">${task}</parameter>
  <parameter name="threadspertask" type="int">${threads}</parameter>
  <parameter name="timelimit">00:10:00</parameter>
  <parameter name="executable">compile/stream.exe</parameter>
  <parameter name="queue">batch</parameter>
  <parameter name="preprocess">${load_modules}; </parameter>
  <parameter name="measurement">time</parameter>
</parameterset>
```

Listing 4.1: Parameterset zur Angabe der nötigen Konfigurationen zum Ausführen der Anwendung

Dieses Parameterset nutzt ein schon vorhandenes, von JUBE vordefiniertes Parameterset aus der Datei platform.xml, das zum Ausführen von Anwendungen über Slurm genutzt werden kann. Mithilfe des Parametersets werden die nötigen Konfigurationen wie Knoten-, Task- und Threadanzahl festgelegt.

Für die Durchführung der Benchmarks wurde einen Rechenknoten von JUSUF genutzt und die Task- und Threadanzahl variiert. Im oberen Code-Beispiel ist zu sehen, dass die Task- und Threadanzahl durch die Parameter `task` und `threads` angegeben werden, die in dem folgenden Parameterset definiert werden:

```
<parameterset name="config">
  <parameter name="mode" type="int">0,1,2</parameter>
  <parameter name="task" type="int" mode="python">['1','4', ",".join([str
      (2**i) for i in range(9)])][$mode]</parameter>
  <parameter name="threads" type="int" mode="python">[",".join([str(2**i)
       for i in range(9)]), ",".join([str(2**i) for i in range(7)]), 1][
      $mode]</parameter>
  <parameter name="cpu_bind">threads</parameter>
  <parameter name="node">block</parameter>
  <parameter name="socket">cyclic</parameter>
  <parameter name="core">fcyclic</parameter>
</parameterset>
```

Listing 4.2: Parameterset zur Angabe der nötigen Konfigurationen der Task- und Threadanzahl

Dieses Parameterset enthält die Konfigurationen der Task- und Threadanzahl und die Bindungs- und Verteilungsoptionen. Um die Task- und Threadanzahl zu konfigurieren, wird der Parameter `mode` genutzt. Mithilfe dieses Hilfsparameters wird die aktuelle Taskanzahl bestimmt. Dafür enthält der Parameter `task` einen Python-Befehl der folgende Liste erstellt: `['1', '4', ['1', '2', '4', '8', '16', '32', '64', '128', '256']]`. Anhand der Taskanzahl kann die Threadanzahl bestimmt werden. Der Parameter `threads` erzeugt ebenfalls mithilfe eines Pythonbefehls folgende Liste: `[['1', '2', '4', '8', '16', '32', '64', '128', '256'],[['1', '2', '4', '8', '16', '32', '64'], '1']`. Somit kann die Konfigurationen der Task- und Threadanzahl in drei Testkombinationen aufgeteilt werden: Einmal wird die Taskanzahl auf eins gesetzt und die Threads in Zweierpotenzschritten erhöht bis zur Threa-

danzahl 256, da ein Rechenknoten auf JUSUF insgesamt 256 Kerne besitzt. Im zweiten Lauf wird die Taskanzahl auf vier erhöht und die Threadanzahl ebenfalls in Zweierpotenz-Schritten bis auf 64 erhöht. Im letzten Lauf wird die Threadanzahl auf eins festgelegt und die Taskanzahl wurde in Zweierpotenz-Schritten bis auf 256 erhöht. Zur besseren Übersicht über die verschiedenen Saklierungen soll folgende Tabelle dienen:

Name	Taskanzahl	Threadanzahl
Thread-Skalierung	1	1, 2, 4, 8, 16, 32, 64, 128, 256
Hybrid-Saklierung	4	1, 2, 4, 8, 16, 32, 64
Task-Skalierung	1, 2, 4, 8, 16, 32, 64, 128, 256	1

Zudem gibt es noch zwei weitere Parametersets:

```
<parameterset name="modules">
  <parameter name="load_modules">module load Intel; module load OpenMPI</
      parameter>
</parameterset>

<parameterset name="executeset" init_with="platform.xml">
  <parameter name="args_starter" separator="|">-n ${tasks} -c ${threads}
      --cpu_bind=verbose,${cpu_bind} --distribution=${node}:${socket}:${
      core} --disable-turbomode</parameter>
</parameterset>
```

Listing 4.3: Parametersets zur Angabe der Module und Argumente zur Ausführung der Anwendung

Das Parameterset `modules` enthält die zu ladendende Module (Pakete die vorinstalliert auf JUSUF zur Verfügung stehen) für den Benchmark und das `executeset`, dass ebenfalls das Slurm Template nutzt, speichert als Parameter einige Argumente zum Ausführen der Anwendung. In diesem Parameter wird auch die jeweilige Konfiguration der Bindungs- und Verteilungsoptionen festgehalten.

Mithilfe des folgenden Filesets wird der Quellcode des STREAM-Benchmarks in das aktuelle Arbeitsverzeichnis geladen.

```
<fileset name="sources">
  <copy>stream_mpi.c</copy>
</fileset>
```

Listing 4.4: Fileset zum kopieren des Quellcodes in das aktuelle Arbeitsverzeichnis

Um die Ergebnisse des Benchmarks auslesen zu können, wurde mithilfe eines Patternsets verschiedene Pattern zum Auslesen der Ergebnisse (3.10) des STREAM-Benchmarks erstellt. Das Patternset ist wie folgt aufgebaut:

```
<patternset name="pattern">
  <pattern name="copy" type="float">Copy:.*?${jube_pat_fp}(?:.*?${
      jube_pat_nfp}){3}</pattern>
  <pattern name="scale" type="float">Scale:.*?${jube_pat_fp}(?:.*?${
      jube_pat_nfp}){3}</pattern>
  <pattern name="add" type="float">Add:.*?${jube_pat_fp}(?:.*?${
      jube_pat_nfp}){3}</pattern>
  <pattern name="triad" type="float">Triad:.*?${jube_pat_fp}(?:.*?${
      jube_pat_nfp}){3}</pattern>
</patternset>
```

Listing 4.5: Patternset mit den Pattern zum Auslesen der Ergebniswerte des STREAM-Benchmarks

Für jede Ergebnisgröße gibt es ein Pattern mit dem entsprechenden Namen, welches einen regulären Ausdruck zum Auslesen des Wertes beinhaltet.

Diese JUBE-Datei enthält zwei Ausführungsschritte, einerseits den Schritt zum Kompilieren der C-Datei und andererseits den Schritt zum Ausführen der daraus resultierenden ausführbaren Datei. Die zwei Schritte sind im Folgenden zu sehen:

```
<step name="compile">
  <use>modules</use>
  <use>sources</use>
  <do>${load_modules}</do>
  <do>mpicc -fopenmp stream_mpi.c -o stream.exe</do>
</step>
<step name="execute" depend="compile" iterations="3">
  <use>executeset</use>
  <use>systemParameter</use>
  <use>config</use>
  <do>${load_modules}</do>
  <do>export OMP_NUM_THREADS=${threads}</do>
  <do>${starter} ${args_starter} ${executable} ${args_exec}</do>
</step>
```

Listing 4.6: Ausführungsschritte zum Kompilieren und Ausführen des STREAM-Benchmarks

Der Schritt `compile` verwendet die Sets für die Module und die benötigte C-Datei. Mithilfe dieser Sets lädt er die Module und kompiliert die C-Datei. Zum kompilieren der C-Datei wurde der GCC-Compiler[2] (Version 10.3) und OpenMPI in der Version 3.1 genutzt. Durch das Kompilieren entsteht die ausführbare Datei `stream.exe`. Dieser Ausführungsschritt wird einmalig ausgeführt.

Der Schritt `execute` ist abhängig vom Schritt `compile` und kann somit erst nach Fertigstellung der Kompilierung beginnen. Dieser Ausführungsschritt benötigt die Inhalte einiger Parametersets und lädt ebenfalls die Module. Da sowohl MPI als auch OpenMP genutzen wird, wird vor der Ausführung die Anzahl der Threads mithilfe des Parameters `OMP_NUM_THREADS` festgelegt. Das letzte `<do>` führt die ausführbare Datei mithilfe von Template-Parametern aus. Um Abweichungen zu vermeiden, wird dieser Schritt dreifach für jede Konfiguration ausgeführt.

Zum Schluss wird eine Ergebnistabelle erzeugt, die die Task- und Threadanzahl, die zugehörige Verteilung und die vier Ergebnisgrößen mit deren jeweiligen Mini-, Maximum und Durchschnitt beinhaltet.

4.1.2. Batch Script

Zum Ausführen der insgesamt sieben JUBE-Eingabedateien wurde ein Batch Skript genutzt, welches wie folgt aussieht:

```
#!/bin/bash -x
#SBATCH --nodes=1
#SBATCH --time=24:00:00
jube run stream_threads.xml
jube run stream_cores.xml
jube run stream_rank.xml
jube run stream_rank_ldom.xml
jube run stream_threads_b_c_b.xml
```

[2] https://gcc.gnu.org/

```
jube run stream_threads_b_c_c.xml
jube run stream_threads_b_f_f.xml
jube run stream_threads_b_b_f.xml
```

Listing 4.7: Batch-Skript zum automatisierten Ausführen aller JUBE-Eingabedateien

Die ersten vier Zeilen des Skripts allokieren einen Knoten. Nach dem Allokieren werden die sieben JUBE-Eingabedateien ausgeführt und für jede Datei ein eigenes Unterverzeichnis erstellt. Innerhalb dieses Verzeichnisses ist jeder Schritt aufgeführt und mithilfe von JUBE-Befehlen können die Ergebnisse eines Laufes ausgeben werden.

4.2. Durchführung des HPL-Benchmarks

Der Quellcode des HPL-Benchmarks ist auf der Internetseite des Netlib Repositorys[3] zu finden. Zusätzlich zum Quellcode sind hier auch die Dokumentation und Ergebnisse einzelner Systeme angegeben. Für die Läufe wurde die neuste Version, Version 2.3 genutzt. Im Gegensatz zum STREAM-Benchmark wurde das Kompilieren des Benchmarks vorab einmal manuell ausgeführt, da das Kompilieren bei diesem Benchmark komplizierter ist. Dabei wurden ebenso GCC als auch OpenMPI verwendet. Als Implementation für BLAS wurde die IntelMKL verwendet. Somit entfällt der Kompilationsschritt und die ausführbare Datei `xhpl` ist bereits gegeben.

Nachfolgend wird die Anpassung der Konfigurationsdatei und die Ausführung der Läufe mithilfe von JUBE und einem Batch Skript beschrieben. Die genutzten Konfigurationen der Bindungs- und Verteilungsoptionen sowie die Skalierungen der Task- und Threadanzahlen wurden ebenfalls wie im Beispiel des STREAM-Benchmarks genutzt und können somit aus den Tabellen in Kapitel 4.1.1 entnommen werden. Da die Ausführung der Läufe auch sonst sehr ähnlich zum STREAM-Benchmark abläuft, wird in den folgenden Unterkapiteln lediglich auf die Unterschiede zum STREAM-Benchmark eingegangen.

4.2.1. Konfigurationsdatei

Zum Ausführen des HPL-Benchmarks wird eine Konfigurationsdatei `HPL.dat` benötigt, die einige Werte und Größen zum Ausführen enthält. Beispielsweise enthält die Konfigurationsdatei die Problemgröße und Größen zur Aufteilung der Daten auf die Prozesse. Die genutzte Konfigurationsdatei für diese Arbeit sieht wie folgt aus:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
32768
1            # of NBs
64
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
1            Ps
#NUMBER#     Qs
```

[3] https://www.netlib.org/benchmark/hpl/

```
16.0          threshold
1             # of panel fact
2             PFACTs (0=left, 1=Crout, 2=Right)
1             # of recursive stopping criterium
4             NBMINs (>= 1)
1             # of panels in recursion
2             NDIVs
1             # of recursive panel fact.
1             RFACTs (0=left, 1=Crout, 2=Right)
1             # of broadcast
1             BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1             # of lookahead depth
0             DEPTHs (>=0)
2             SWAP (0=bin-exch,1=long,2=mix)
64            swapping threshold
0             L1 in (0=transposed,1=no-transposed) form
0             U  in (0=transposed,1=no-transposed) form
1             Equilibration (0=no,1=yes)
8             memory alignment in double (> 0)
```

Listing 4.8: Beispiel einer Konfigurationsdatei `HPL.dat` des HPL-Benchmarks

Die ersten zwei Zeilen sind als Überschrift der Datei anzusehen und spielen für das Ausführen des Benchmarks keine Rolle. In Zeile drei kann das Ausgabemedium bestimmt werden, welches für diese Arbeit die Standard-, also die Konsolenausgabe darstellt. Die zwei folgenden Zeilen beschreiben die Problemgröße `N`. Dabei wurde ebenfalls eine Zweierpotenz gewählt, damit diese sowohl durch die Blockgröße NB, die in Zeile sieben und acht festgelegt wird, teilbar ist und diese wiederum auf die Anzahl Task bzw. Threads aufgeteilt werden kann, die ebenfalls Zweierpotenzen entsprechen. Außerdem wurde die Problemgröße so groß gewählt, um den L3-Cache des HPC Systems JUSUF auszureizen. Mithilfe der Zeilen zehn bis zwölf kann die Größe des P-mal-Q-Rasters angegeben werden. Dabei ist zu beachten, dass das Produkt von P und Q der Anzahl der genutzten Prozessen entsprechen muss. Somit wurde P auf eins gesetzt und das Q mithilfe eines Substitutionssets, dass den Wert `#NUMBER#` durch die Prozessanzahl ersetzt, zu ergänzen. Dies entspricht keiner optimalen Aufteilung für den HPL-Benchmark, da aber vornehmlich mehrerer Läufe mit gleichem Setup verglichen werden sollen, ist diese Einschränkung ausreichend. Die restlichen Zeilen beschreiben weitere Einstellmöglichkeiten, die jedoch auf den Standardeinstellungen verblieben sind und für die Fragestellung dieser Arbeit nicht relevant sind.

4.2.2. JUBE-Implementierung

Um die externen Dateien `xhpl` und `HPL.dat` in das Arbeitsverzeichnis jedes Laufes zu kopieren wurde ein Fileset genutzt. Da jedoch der Parameter Q der Datei `HPL.dat` noch angepasst werden muss, wurde ein Substitutionsset erstellt.

```
<substituteset name="substitute">
  <iofile in="HPL.dat" out="HPL.dat" />
  <sub source="#NUMBER#" dest="$task" />
</substituteset>
```

Listing 4.9: Substitutionsset zum Anpassen der Konfigurationsdatei `HPL.dat` des HPL-Benchmarks

Mithilfe dieses Substitutionssets kann der Platzhalter `#NUMBER#` in der Konfigurationsdatei an die aktuelle Taskanzahl angepasst werden.

Die Pattern zum Auslesen der Ergebnisgrößen unterscheiden sich selbstverständlich von denen des STREAM-Benchmarks und sind wie folgt aufgebaut:

```
1 <patternset name="pattern">
2   <pattern name="time" type="float">Gflops.[-]+.(?:\S+\s+){5}${
        jube_pat_fp}</pattern>
3   <pattern name="gflops" type="float">Gflops.[-]+.(?:\S+\s+){6}${
        jube_pat_fp}</pattern>
4 </patternset>
```

Listing 4.10: Patternset mit den Pattern zum Auslesen der Ergebniswerte des HPL-Benchmarks

Es existiert ein Pattern zum Auslesen der benötigten Laufzeit (`time`) und einen für die erreichten Gflops (`gflops`).

Im Gegensatz zum STREAM-Benchmark gibt es nur den Ausführungsschritt `execute`, der die Datei mit verschiedenen Konfigurationen ausführt. Der Kompilationsschritt fällt weg, da der Quellcode bereits manuel kompiliert wurde.

Die Ergebnistabelle des HPL-Benchmarks enthält ebenfalls die Konfigurationen der Task- und Threadanzahl, die Pinningoptionen und die jeweiligen Angaben zu der gebrauchten Zeit und erreichten Gflops.

4.2.3. Batch Skript

Das Batch Skript des HPL-Benchmarks konnte aus dem STREAM-Benchmark übernommen werden.

5. Leistungsvergleich unterschiedlicher Prozess-Pinning Strategien

Um die Leistung der Pinning-Strategien zu vergleichen, wurden mehrere Skalierungsläufe bezüglich der Task- und Threadanzahl mit verschiedenen Pinningoptionen (beide Tabellen sind in Kapitel 4.1.1 zu finden) durchgeführt. Diese Skalierungsläufe lassen sich in drei Konifurationen einteilen, bei der einerseits die Taskanzahl auf eins gesetzt wurde und die Threadanzahl erhöht wurde (Thread-Skalierung) und andererseits die Taskanzahl in der zweiten Konfiguration auf vier gesetzt und die Threadanzahl erneut erhöht sobald alle Speicherknoten einmal belegt wurde (Hybrid-Skalierung). Bei der letzten Konfiguration handelt es sich um eine Task-Skalierung bei der die Threadanzahl auf eins gesetzt wurde und die Taskanzahl erhöht wurde. Die Ergebnisse der verschiedenen Skalierungsläufe wurden in einer Tabelle gesammelt. Zum Veranschaulichen der Ergebnisse wurden Graphen, die den jeweiligen Ergebniswert gegenüber der Anzahl Threads bzw. Tasks aufzeigen, genutzt. Diese Graphen werden in den folgenden Unterkapitel aufgezeigt und analysiert.

5.1. Leistungsvergleich des STREAM-Benchmarks

In Kapitel 3.2.1 wurden die Ergebnisgrößen des STREAM-Benchmarks vorgestellt. Für den Leistungsvergleich dieser Arbeit wurde der Durchschnitt der besten gemessenen Speicherbandbreiten in MB/s von drei STREAM-Benchmarkläufen genutzt. Die STREAM-Benchmarkläufe führten dabei jeweils zehn Messungen durch. Aus diesen zehn Messungen wurde für jeden Lauf die beste gemessene Speicherbandbreite ermittelt und aus den daraus resultierenden Speicherbandbreiten der Durchschnitt ermittelt. Da die durchschnittliche Standardabweichung der Messungen ungefähr 450 MB/s beträgt, was bei Werten von durchschnittlich ca 100.000 MB/s zu eine ziemlich geringen Variation von ca. 0,45 % führt, kann davon ausgegangen werden, dass die durchschnittlichen Werte reproduzierbar sind. Der Durchschnitt der Messungen und die Standardabweichung wurden über das JUBE-Pattern `_avg` bzw. `_std` berechnet.

Im Folgenden werden die drei verschiedenen Konfigurationsarten der Task- und Threadanzahl erneut vorgestellt, die Ergebnisse der verschiedenen Pinning-Optionen miteinander verglichen und diese analysiert.

5.1.1. Konfiguration: Thread-Skalierung

In dieser Konfiguration wurde die Taskanzahl auf eins festgelegt und die Threadanzahl bis zur Gesamtanzahl aller möglichen logischen Kerne eines Knotens (256) erhöht. In der folgenden Abbildung werden die Ergebnisse der verschiedenen Bindungsoptionen gezeigt:

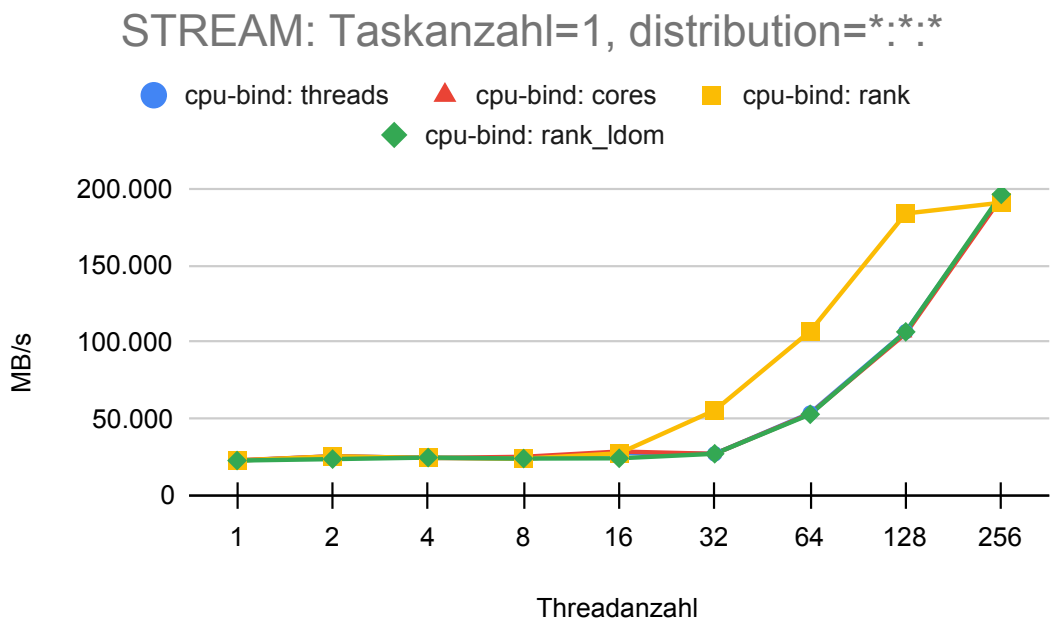
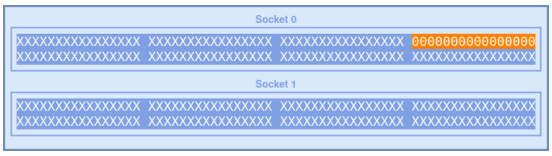
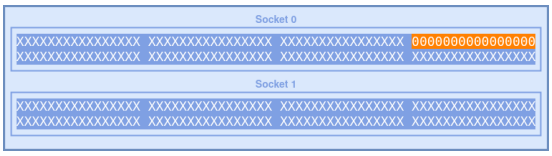


Abbildung 5.1.: Erreichte Bandbreite des STREAM-Benchmarks für die Skalierung der Threadanzahl für verschiedene Bindungsoptionen bei gesetzter Standardverteilung

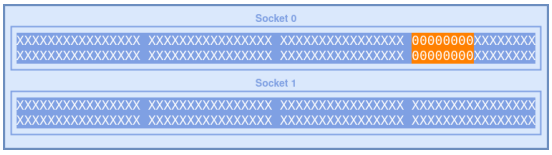
Abbildung 5.1 zeigt das Verhältnis der gemessenen Bandbreite in MB/s anhand der Anzahl genutzter Threads. Die vier Kurven stehen dabei für die vier verschiedenen Bindungsoptionen `threads`, `cores`, `rank` und `rank_ldom`. Es ist zu sehen, dass die Kurven der Bindungsoption bis zu einer Anzahl von 16 Threads nah beieinanderliegen und relativ konstant bleiben. Die folgenden Grafiken sollen veranschaulichen, aus welchem Grund die Werte so nah beieinanderliegen, indem das Pinning der verschiedenen Bindungsoptionen aufgezeigt wird:



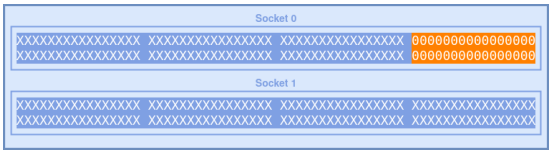
(a) `threads`



(b) `rank`



(c) `rank_ldom`

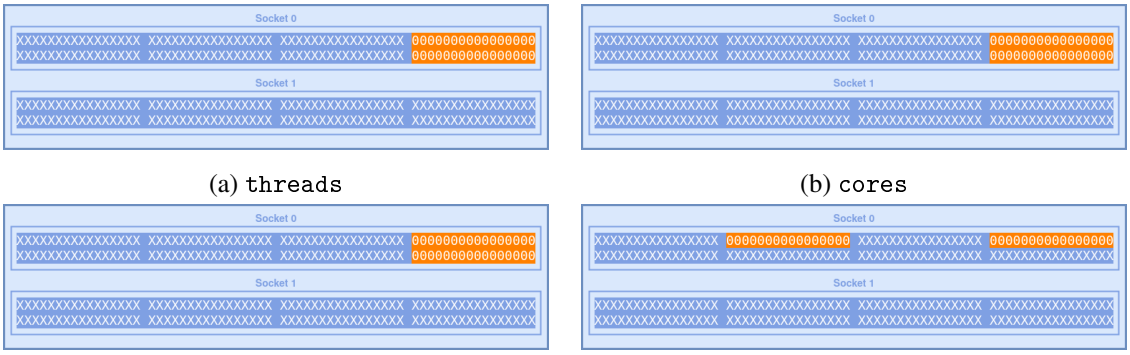


(d) `cores`

Abbildung 5.2.: Pinning-Verhalten der unterschiedlichen Bindungsoptionen anhand von einer Task mit 16 Threads

Die Grafiken der Abbildung 5.2 zeigen das Pinning mithilfe der jeweiligen Bindungsoption von einer Task mit 16 Threads. Es ist zu sehen, dass die Bindungsoptionen `threads` und `rank` die Threads gleichermaßen auf die verfügbaren Kerne verteilen. Somit liegen die Werte der gemessenen Bandbreite für diese beiden Bindungsoptionen nah beieinander. Anhand des Pinning-Verhaltens der Option `rank_ldom` kann darauf geschloßen werden, dass die Nutzung der virtuellen Kerne für diesen Benchmark keinen großen Unterschied im Gegensatz zu den physischen Kerne, da bei dieser Option gleichermaßen physische und virtuelle Kerne belegt werden, jedoch ebenfalls ähnliche Werte gegenüber dem Pinning-Verhalten der Optionen `threads` und `rank` gemessen werden können, die ausschließlich physische Kerne nutzen. Dies lässt sich damit begründen, dass bei diesem Benchmark die Speicheranbindung deutlich wichtiger ist, während die Gleitkommaarithmetik einen geringeren Einfluss auf die Messungen hat. Zudem ist zu sehen, dass die Bindungsoption `cores` das Pinning-Verhalten der Option `threads` übernimmt, aber zusätzlich die virtuellen Kerne belegt. Auch hier werden nur 16 Threads genutzt, sodass das Betriebssystem dynamisch in der Gruppe der 32 gewählten Kerne wählt, dabei jedoch keine bessere Bandbreite erreicht.

Ab 32 Threads ist zu sehen, dass die gemessene Bandbreite der Bindungsoption `rank` deutlich höher ist als die anderen drei Optionen.



(a) `threads` (b) `cores`

(c) `rank_ldom` (d) `rank`

Abbildung 5.3.: Pinning-Verhalten der unterschiedlichen Bindungsoptionen anhand von einer Task mit 32 Threads

Anhand der Beschreibung in Kapitel 2.2.1 und den oben gezeigten Grafiken der Abbildung 5.3, lässt sich dies ebenfalls durch das Pinning-Verhalten erklären. Während die restlichen drei Bindungsoptionen zunächst alle Kerne der vierten NUMA-Domain belegen, belegt die Bindungsoption `rank` (Abbildung 5.3d) nur die physischen Kerne der vierten NUMA-Domain und fährt bei den physischen Kernen der nächsten NUMA-Domain (NUMA-Domain 2, gegeben durch die gesonderte NUMA-Domain-Reihenfolge) fort. Daraus lässt sich schließen, dass das Verteilen der Threads auf mehrere Speicherknoten effizienter ist, da die Threads ihre Berechnungen unabhängig voneinander durchführen und somit nicht auf einen gemeinsamen Speicher zugreifen müssen. Stattdessen profitieren diese von der besseren Speicheranbindung bei Nutzung mehrere NUMA-Domains. Zudem lassen sich aus diesem Pinning-Verhalten auch die konstanten Werte bis 32 Threads für die drei Bindungsoptionen `threads`, `cores` und `rank_ldom` erklären. Da diese Bindungsoptionen bis zu 32 Threads nur eine NUMA-Domain nut-

zen, wird ersichtlich, dass sich die Leistung nicht verschlechtert, desto 'voller' eine NUMA-Domain ist, sondern konstant bleibt. Mit steigender Threadanzahl nutzen die drei Bindungsoptionen ebenfalls mehr NUMA-Domains und holen im Sinne der verfügbaren Speicherbandbreite zur Bindungsoption `rank` auf.

Zusätzlich zu den vier verschiedenen Bindungsoptionen wurden außerdem verschiedene Verteilungsoptionen getestet. Die folgende Grafik zeigt die Ergebnisse der gemessenen Bandbreite in MB/s in Abhängigkeit zu der genutzten Threadanzahl:

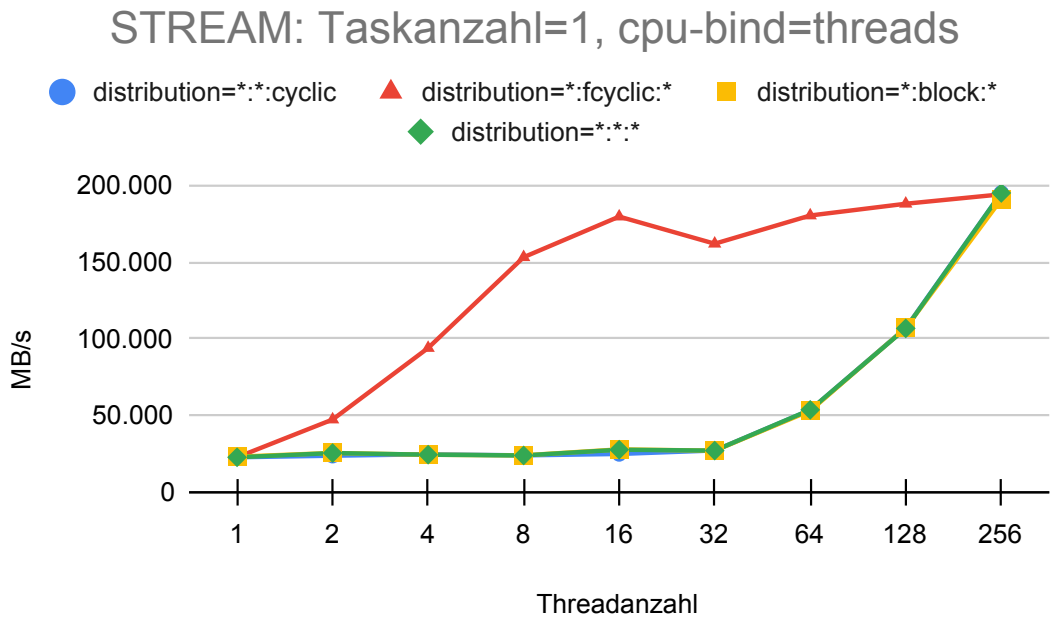
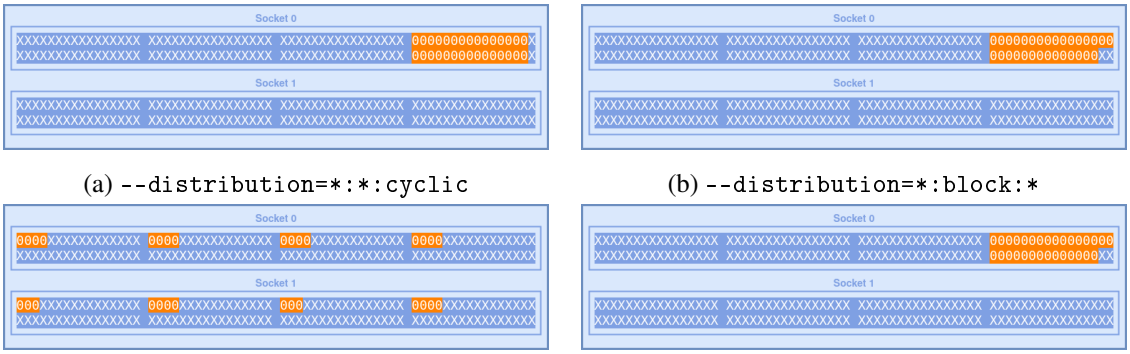


Abbildung 5.4.: Erreichte Bandbreite des STREAM-Benchmarks für die Skalierung der Threadanzahl für verschiedene Verteilungsoptionen bei gesetzter Standardbindung

Auch in dieser Grafik ist zu sehen, dass die Verteilungsoptionen `--distribution=*:*:cyclic`, `--distribution=*:block:*`, wie auch die Standardverteilungsoption `--distribution=*:*:*` ähnliche und vor allem konstante Bandbreitenwerte bis zu 32 Threads aufweisen, während die Option `--distribution=*:fcyclic:*` deutlich höhere Werte vermerkt. Die folgenden Abbildungen zeigen das Pinning-Verhalten der vier Verteilungsoptionen anhand von einer Task mit 30 Threads:



(a) `--distribution=*:*:cyclic` (b) `--distribution=*:block:*`

(c) `--distribution=*:fcyclic:*` (d) `--distribution=*:*:*`

Abbildung 5.5.: Pinning-Verhalten der unterschiedlichen Verteilungsoptionen anhand von einer Task mit 30 Threads

Anhand der Abbildungen 5.5 und der Erklärung in Kapitel 2.2.2 ist zu erkennen, dass sich das Pinning-Verhalten der Optionen `--distribution=*:*:cyclic` und `--distribution=*:block:*` sich lediglich darin unterscheiden, dass die Option `--distribution=*:*:cyclic` sowohl die physischen als auch die virtuellen Kerne nacheinander besetzt, während die Option `--distribution=*:block:*` zuerst die physischen und dann die virtuellen Kerne belegt, wie auch die Standardverteilungsoption. Wie jedoch festgestellt wurde, entsteht kein erheblicher Leistungsunterschied zwischen dem Nutzen der physischen bzw. virtuellen Kerne. Außerdem belegen die beiden Optionen bis zu 32 Threads nur eine NUMA-Domain, was für die konstanten Werte sorgt. Die Option `--distribution=*:fcyclic:*` verteilt jedoch die Threads auf alle NUMA-Domains, sodass eine höhere Bandbreite gemessen werden kann. Allerdings ist hier auch zu bemerken, dass bereits ab acht Threads alle NUMA-Domains belegt sind und kein größerer Anstieg der Bandbreite mehr erreicht werden kann.

5.1.2. Konfiguration: Hybride Skalierung

In dieser Konfiguration wurde die Taskanzahl auf vier festgelegt und die Threadanzahl erhöht. Die folgende Abbildung zeigt die gemessenen Bandbreitenwerte anhand der Threadanzahl mit verschiedenen Bindungsoptionen:

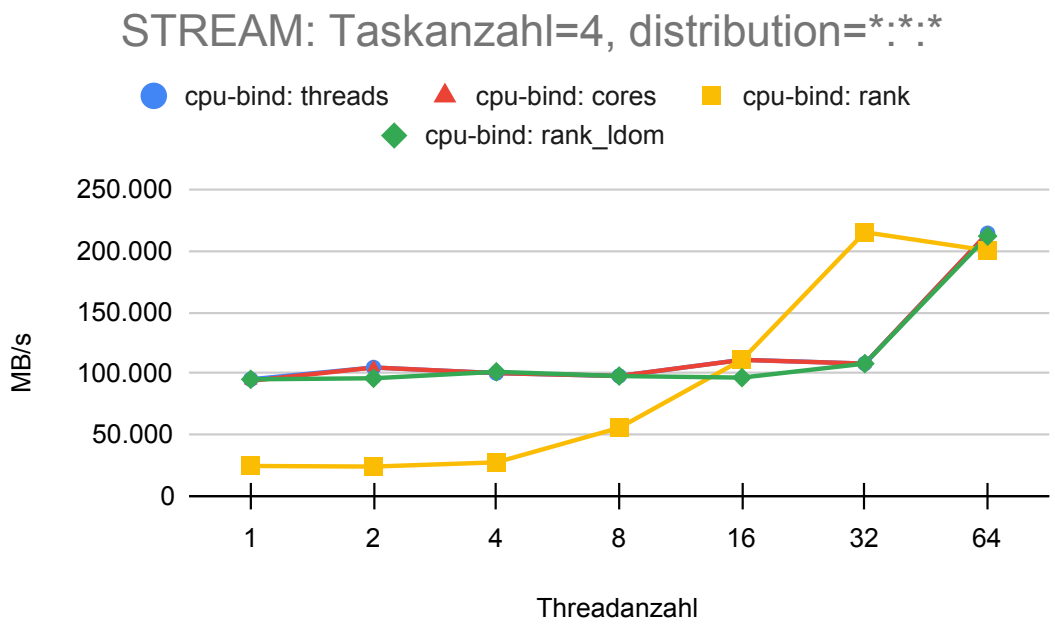
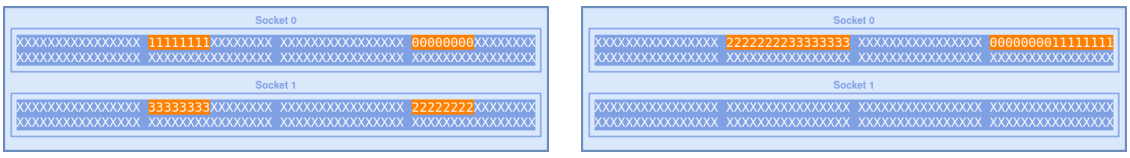


Abbildung 5.6.: Erreichte Bandbreite des STREAM-Benchmarks für vier Tasks und der Skalierung der Threadanzahl für verschiedene Bindungsoptionen bei gesetzter Standardverteilung

Aufgrund des ähnlichen Pinning-Verhaltens liegen auch hier die Bandbreitenwerte der Bindungsoptionen `threads`, `cores` und `rank_ldom` nah beieinander, während die Bindungsoption `rank` bis zu 16 Threads deutlich niedrigere, jedoch ab 32 Threads deutlich höhere Bandbreitenwerte aufweist. Die folgenden Grafiken sollen den Wechsel erklären. Zunächst wird das Pinning-Verhalten der Bindungsoptionen `threads` und `rank` anhand von vier Tasks mit je acht Threads gegenübergestellt:

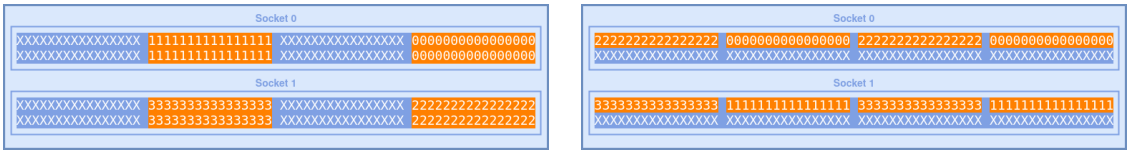


(a) `threads` (b) `rank`

Abbildung 5.7.: Pinning-Verhalten der unterschiedlichen Bindungsoptionen anhand von vier Task mit je acht Threads

In den Abbildungen 5.7 ist zu erkennen, dass die Bindungsoption `threads` in dieser Konstellation die Tasks über mehrere NUMA-Domains verteilt, während die Bindungsoption `rank` lediglich zwei NUMA-Domains ausnutzt. Dies erklärt die deutlich niedrigeren Bandbreitenwerte bei einer Threadanzahl bis zu 16, weil bis zu dieser Threadanzahl bei vier Tasks genau zwei NUMA-Domains ausreichen. Außerdem werden bei bis zu 32 Threads pro Task lediglich diese vier NUMA-Domains besetzt, was die konstanten Bandbreitenwerte bei den anderen Bindungsoptionen erklärt. Bei genau 16 Threads

ist das Pinning-Verhalten aller vier Bindungsoptionen erneut nahezu identisch, sodass die Bandbreitenwerte für diese Konfiguration für alle Optionen nur geringe Unterschiede zeigen. Folgend wird das Pinning-Verhalten der beiden Bindungsoptionen mit vier Tasks und 32 Threads gezeigt, um zu verdeutlichen, weshalb die Bindungsoption `rank` ab diesem Wert bessere Bandbreitenwerte liefert:



(a) `threads` (b) `rank`

Abbildung 5.8.: Pinning-Verhalten der unterschiedlichen Bindungsoptionen anhand von vier Tasks mit je 32 Threads

Während die Abbildung 5.8a zeigt, dass die Bindungsoption `threads` alle Kerne der vier NUMA-Domains belegt, ist in Abbildung 5.8b zu sehen, dass die Bindungsoption `rank` die physischen Kerne aller NUMA-Domains belegt und somit höhere Bandbreitenwerte zu messen sind.

Die nächste Grafik zeigt die Ergebnisse der Bandbreitenmessung anhand von verschiedenen Verteilungsoptionen der Bindungsoption `threads`:

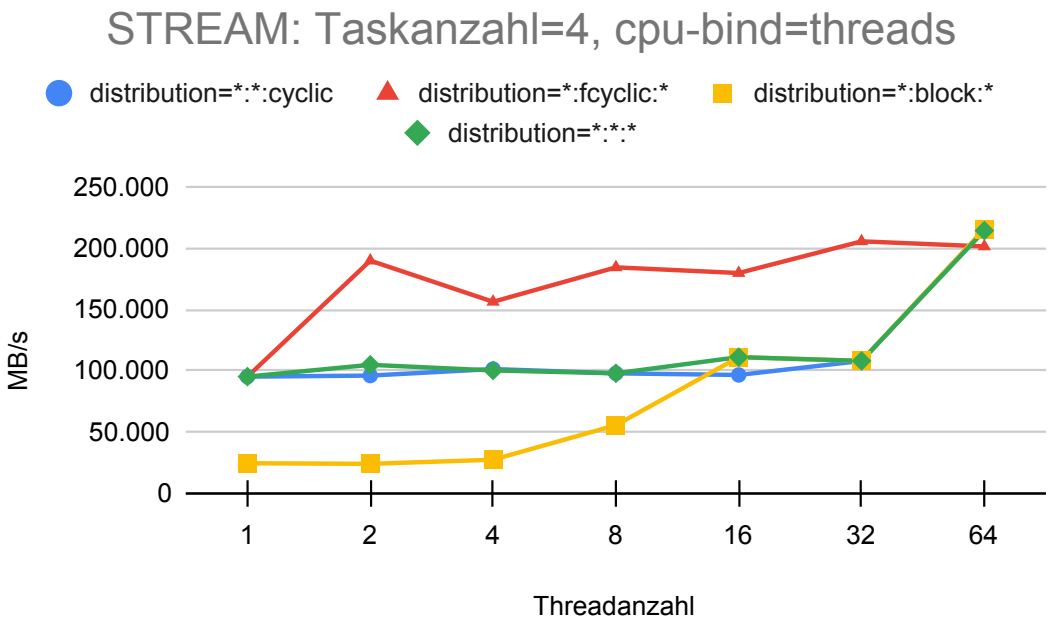


Abbildung 5.9.: Erreichte Bandbreite des STREAM-Benchmarks für vier Tasks und der Skalierung der Threadanzahl für verschiedene Verteilungsoptionen bei gesetzter Standardbindung

Die Abbildung 5.9 zeigt, dass die Verteilungsoption `--distribution=*:*:cyclic` einen sehr ähnlichen Verlauf zur Standardverteilungsoption besitzt, während die Option

`--distribution=*:block:*` deutlich schlechtere und `--distribution=*:fcyclic:-*` deutlich bessere Werte für kleinere Threadanzahlen liefert. Außerdem ist zu sehen, dass die Bandbreitenwerte für die Option `--distribution=*:fcyclic:*` ab zwei Threads relativ konstant bleiben. Die folgenden Grafiken zeigen das Pinning-Verhalten der Optionen anhand von vier Tasks mit je vier Threads:

(a) `--distribution=*:*:cyclic`

(b) `--distribution=*:*:*`

(c) `--distribution=*:fcyclic:*`

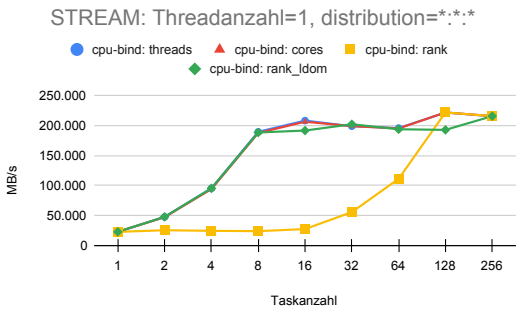
(d) `--distribution=*:block:*`

Abbildung 5.10.: Pinning-Verhalten der unterschiedlichen Verteilungsoptionen anhand von vier Tasks mit je vier Threads

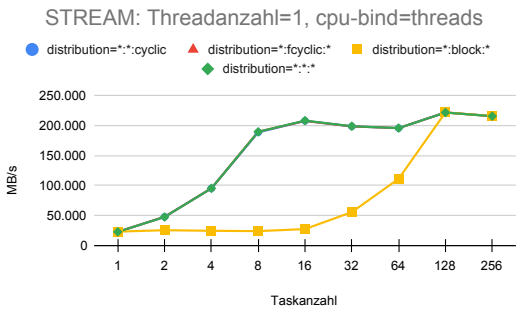
Es ist erneut zu sehen, dass das Ausnutzen der virtuellen statt der physischen Kerne keinen Leistungsunterschied ergibt, jedoch das Verteilen der Tasks und Threads auf mehrere NUMA-Domains die Bandbreitenwerte erhöht. Zudem zeigt sich, dass die Werte für die Option `--distribution=*:fcyclic:*` konstant bleiben, da bereits ab zwei Threads die NUMA-Domains gleichmäßig belegt werden.

5.1.3. Konfiguration: Task-Skalierung

In dieser Konfiguration wurde die Threadanzahl auf eins festgelegt und die Taskanzahl erhöht. In den folgenden Abbildungen werden die Ergebnisse der verschiedenen Konfigurationen der Bindungs- und Verteilungsoptionen dargestellt:



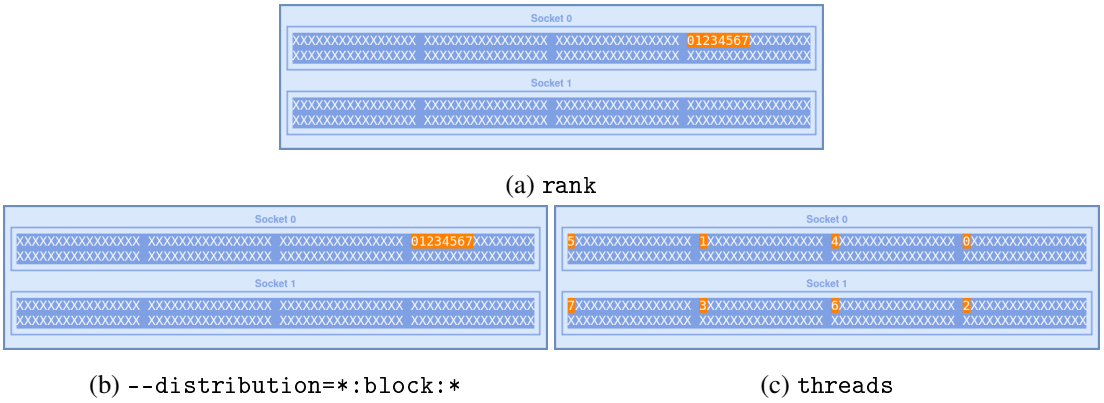
(a) Verschiedene Bindungsoptionen bei gesezter Standardverteilung



(b) Verschiedene Verteilungsoptionen bei gesetzter Standardbindung

Abbildung 5.11.: Erreichte Bandbreite des STREAM-Benchmarks für einen Thread und der Skalierung der Taskanzahl

Wie in den vorherigen Ergebnisabbildungen ist auch hier zu sehen, dass die Ergebniswerte vieler Konfigurationen nah beieinanderliegen, während die Bandbreitenwerte der Bindungsoption `rank` und der Verteilungsoption `--distribution=*:block:*` in Abbildung 5.11 stark abweichen. Im Gegensatz zu den vorherigen Ergebnissen, ist es in diesem Fall so, dass die Bandbreitenwerte dauerhaft deutlich niedriger sind. Anhand von acht Tasks mit je einem Thread wird im Folgenden analysiert, wie es zu den abweichenden Werten kommt. Die folgenden Grafiken zeigen das Pinning-Verhalten mit dieser Konfiguration der Bindungsoptionen `threads` und `rank` und der Verteilungsoption `--distribution=*:block:*`.



(a) `rank`

(b) `--distribution=*:block:*`

(c) `threads`

Abbildung 5.12.: Pinning-Verhalten der unterschiedlichen Verteilungs- und Bindungsoptionen anhand von vier Tasks mit je acht Threads

Wie in Abbildung 5.12c zu sehen ist, verteilt die Bindungsoption `threads` die Tasks gleichmäßig auf alle NUMA-Domains und nutzt somit auch eine hohe Speicherbandbreite. Die Bindungsoption `rank` und die Verteilungsoption `--distribution=*:block:*` belegen eine NUMA-Domain mit allen Tasks, sodass nur eine NUMA-Domain besetzt ist. Dies erklärt die deutlich niedrigeren Bandbreitenwerte. Ebenfalls ist zu sehen, dass die Bindungsoption `threads` ab acht Threads alle NUMA-Domains belegt, wodurch die konstanten Werte ab acht Threads zu erklären sind.

5.1.4. Fazit des STREAM-Benchmarks

Abschließend und allgemein zu allen Grafiken ist noch zu erwähnen, dass die Bandbreitenwerte bei voller Auslastung des Knotens nahezu identisch sind. Dies gilt für jegliche Konfigurationen, was sich dadurch erklären lässt, dass bei dieser Konfiguration alle Kerne des Knotens voll besetzt sind und das Pinning-Verhalten für die Auslastung keinen Unterschied darstellt. Daneben wird deutlich, wenn ein Knoten nicht voll genutzt wird, dass insbesondere die Bindungsoptionen `rank` für hybride bzw. reine Threadanwendungen zu besseren Speicherbandbreitenwerten führen kann (bei hybriden Anwendungen in Abhängigkeit der Threadanzahl), während diese Bindungsoption bei einer reinen Taskanwendung für den STREAM-Benchmark, bei der die Speichbandbreite dominiert, vermieden werden sollte. Außerdem ist zu erkennen, dass die Task-Skalierung für das gewählte STREAM-Setup im Allgemeinen bessere Bandbreitenwerte ergibt und das Maximum der Bandbreitenwerte aller Konfigurationen bei 128 Tasks mit einem Thread,

der Bindungsoption `threads` und der Verteilungsoption `--distribution=*:block:*` mit 221437,467 MB/s liegt.

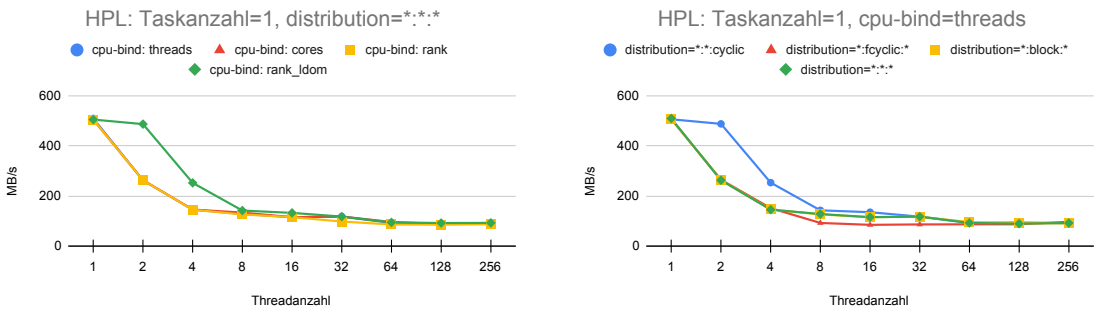
5.2. Leistungsvergleich des HPL-Benchmarks

Der Algorithmus und die Ergebnisgrößen des HPL-Benchmarks wurden bereits in Kapitel 3.2.2 vorgestellt. Für das Analysieren der Ergebnisse wurde die gemessene Zeit genutzt, sodass, im Gegensatz zum STREAM-Benchmark, bei dem die gemessene Bandbreite genutzt wurde, kleinere Werte für ein besseres Ergebnis sprechen (die Laufzeit korreliert dabei immer, mit der möglichen Gleitkommaoperationen pro Sekunde). Jeder Lauf bestand dabei aus drei Messungen, dessen Durchschnitt für die Auswertung genutzt wurde. Bei durchschnittlichen Ergebnissen aller Werte von ca. 130 Sekunden und einer durchschnittlichen Standardabwichung von 1,5 Sekunden, kann bei einer Varianz von 1,15 % von zuverlässlichen Werten ausgegangen werden. Der Durchschnitt und die Standardabweichung wurde mithilfe der JUBE-Pattern `_avg` und `_std` berechnet.

Nachfolgend werden die drei verschiedenen Konfigurationsarten der Task- und Threadanzahl vorgestellt, die Ergebnisse der verschiedenen Pinning-Optionen miteinander verglichen und diese analysiert.

5.2.1. Konfiguration: Thread-Skalierung

In diesem Lauf wurde die Taskanzahl auf eins festgelegt und die Threadanzahl erhöht. In den folgenden zwei Abbildungen, wird die benötigte Zeit in Sekunden zum Durchlaufen des HPL-Benchmarks der verschiedenen Pinning-Optionen gezeigt.

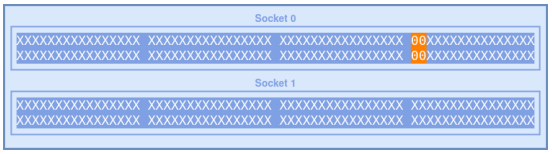


(a) Verschiedene Bindungsoptionen bei gesezter Standardverteilung

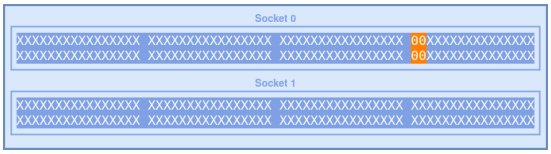
(b) Verschiedene Verteilungsoptionen bei gesetzter Standardbindung

Abbildung 5.13.: Durchschnittliche Laufzeit des HPL-Benchmarks für die Skalierung der Threadanzahl

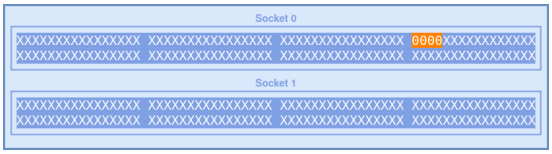
In den Grafiken der Abbildung 5.18a ist zu sehen, dass alle Konfigurationen der Bindungs- und Verteilungsoptionen ähnliche Zeiten ausweisen, bis auf die Bindungsoption `rank_ldom` (Abbildung 5.13a) und die Verteilungsoption `--distribution=*:*:cyclic` (Abbildung 5.13b), die zwischen zwei und acht Threads ein schlechteres Ergebnis liefern. Anhand der folgenden Grafik soll das Pinning-Verhalten der beiden Optionen im Vergleich zum Standard Pinning-Verhalten aufgezeigt werden:



(a) threads, `--distribution=*:*:cyclic`



(b) `rank_ldom`



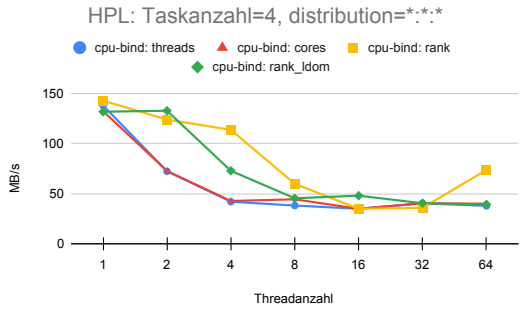
(c) `threads`, `--distribution=*:*:*`

Abbildung 5.14.: Pinning-Verhalten der unterschiedlichen Verteilungs- und Bindungsoptionen anhand von einer Task mit je vier Threads

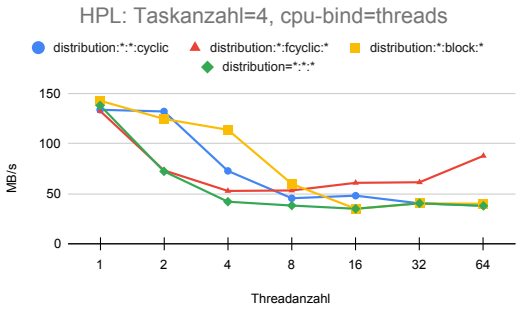
Die Abbildungen zeigen das Pinning bei einem Task mit vier Threads und der jeweiligen Konfiguration der Bindungs- bzw. Verteilungsoption. Es ist zu sehen, dass die beiden Optionen `--distribution=*:*:cyclic` und `rank_ldom` das gleiche Pinning aufweisen und sich zu der Bindungsoption `threads` lediglich durch die Nutzung der virtuellen Kerne unterscheidet. Somit lässt sich darauf schließen, dass im Gegensatz zum STREAM-Benchmark, die reine Nutzung der physischen Kerne ein besseres Ergebnis liefert. Dies lässt sich auf die größere Nutzung komplexerer Gleitkommaoperationen in HPL zurückführen.

5.2.2. Konfiguration: Hybrid

In diesem Lauf wurde die Taskanzahl auf vier festgelegt und die Threadanzahl erhöht. In den folgenden zwei Abbildungen werden die Ergebnisse der verschiedenen Pinning-Optionen gezeigt.



(a) Verschiedene Bindungsoptionen bei gesezter Standardverteilung



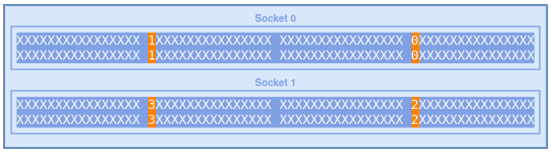
(b) Verschiedene Verteilungsoptionen bei gesetzter Standardbindung

Abbildung 5.15.: Durchschnittliche Laufzeit des HPL-Benchmarks für vier Tasks und der Skalierung der Threadanzahl

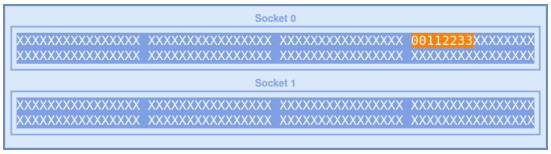
Abbildung 5.15 zeigt, dass die Bindungsoptionen `threads`, `cores` (Abbildung 5.15a) und die Standartverteilungsoption `--distribution=*:*:*` (Abbildung 5.15b) ähnli-

che Werte für die gemessene Zeit liefern, jedoch die Bindungsoptionen `rank`, `rank_ldom` (Abbildung 5.15a) und die Verteilunsoptionen `--distribution=*:*:cyclic` und `--distribution=*:block:*` (Abbildung 5.15b) deutlich schlechtere Werte zurückgeben. Bei genauerer Betrachtung ist zu erkennen, dass die Ergebnisse der Bindungsoption `rank` und der Verteilungsoption `--distribution=*:block:*` ebenfalls nah beieinanderliegen, genauso wie die Bindungsoption `rank_ldom` und die Verteilungsoption `--distribution=*:*:cyclic`. Dies lässt sich durch dasselbe Pinning-Verhalten der jeweiligen Konfigurationen beschreiben.

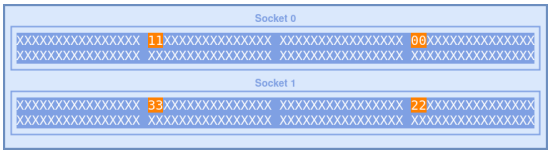
Die folgenden Grafiken zeigen das Pinning-Verhalten der verschiedenen Bindungs- und Verteilungsoptionen:



(a) `rank_ldom` und `--distribution=*:*:cyclic`



(b) `rank` und `--distribution=*:block:*`



(c) `threads`

Abbildung 5.16.: Pinning-Verhalten der unterschiedlichen Verteilungs- und Bindungsoptionen anhand von vier Tasks mit je zwei Threads

Abbildung 5.16a zeigt, dass die Threads, im Gegensatz zur Bindungsoption `threads` (Abbildung 5.16c), auch an die virtuellen Kerne gebunden werden, was die schlechteren Ergebnisse der Bindungsoption `rank_ldom` und der Verteilungsoption `--distribution=*:*:cyclic` erklären. Außerdem ist in Abbildung 5.16b zu erkennen, dass alle Tasks und Threads lediglich auf eine NUMA-Domain gebunden wurden, sodass durch die langsameren Ergebnisse darauf geschloßen werden kann, dass das Verteilen auf mehrere NUMA-Domains zu besseren Ergebnissen führt. Somit sind die Bindungsoption `rank` und die Verteilungsoption `--distribution=*:block:*` durch ihre NUMA-Domain-Verteilung langsamer, als die Standardbindung `threads`.

Außerdem ist in der Abbildung 5.15b zu sehen, dass die Verteilung `--distribution=*:fcyclic:*` für kleinere Threadanzahlen ähnliche Werte zur Standardverteilung `--distribution=*:*:*` aufweist, jedoch ab vier Threads pro Task kontinuierlich langsamer wird. Die folgende Grafik zeigt das Pinning-Verhalten der Verteilungsoption `--distribution=*:fcyclic:*` mit jeweils vier bzw. 16 Threads und vier Tasks:

(a) Vier Threads pro Task

(b) 16 Threads pro Task

Abbildung 5.17.: Pinning-Verhalten der Verteilungsoptionen `--distribution=*:fcyclic:*` mit jeweils vier Tasks und unterschiedlichen Threadanzahlen

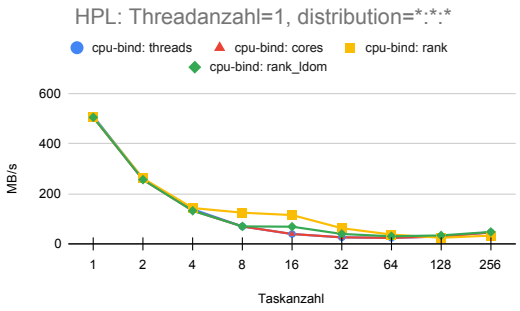
Beim Ausführen einer Anwendung wird vom Prozessor ein lokaler Speicher angelegt, der typischerweise auf der NUMA-Domain des ersten Threads liegt. Da die Threads des HPL-Benchmarks nicht unabhängig voneinander laufen und die Threads somit auf diesen lokalen Speicher zugreifen müssen, kann es zu kostspieligen Remote-Zugriffen kommen, wenn der zugreifende Thread auf einer anderen NUMA-Domain liegt. In Abbildung 5.17a ist zu sehen, dass die Threads bereits auf verschiedenen NUMA-Domains verteilt sind und es somit zu Remote-Zugriffen kommt. Allerdings sind die Threads einer Task auf nur zwei verschiedene NUMA-Domains verteilt. Im Fall von 16 Threads in Abbildung 5.17b ist zu erkennen, dass auf jeder NUMA-Domain mindestens ein Thread jeder Task liegt und somit zu Remot-Zugriffen über alle NUMA-Domains kommt, was für längere Laufzeit sorgt.

Anhand dieser Leistungsverschlechterung lässt sich auch die langsameren Laufzeiten eines voll belegten Knotens bei der Bindungsoption `rank` und der Verteilungsoption `--distribution=*:fcyclic:*` im Gegensatz zu den anderen Konfigurationen erklären. Während bei allen anderen Konfigurationen eine Task mit ihren 64 Threads genau zwei NUMA-Domains belegt, sind bei diesen Konfigurationen jeweils zwei (`rank`) bzw. vier (`--distribution=*:fcyclic:*`) Tasks auf einer NUMA-Domain zu finden, sodass mehr Zeit für die Speicherzugriffe der Threads benötigt wird.

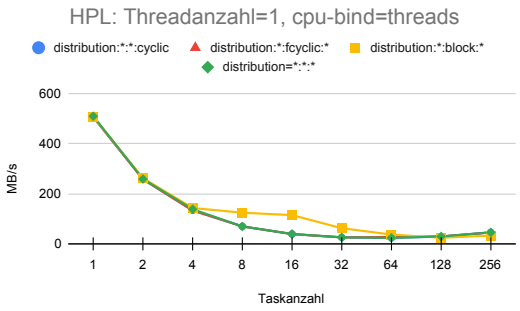
Abschließend ist noch zu der hybriden Konfiguration zu sagen, dass diese deutlich schneller als die anderen Konfigurationen ist. Während die anderen Konfigurationen bei einer Laufzeit von ca. 500 Sekunden für geringe Threadanzahl starten, beginnt diese bereits bei ca. 140 Sekunden. Das lässt sich auf den Einsatz von sowohl MPI als auch OpenMP und die gewählte Problemgröße zurückführen.

5.2.3. Konfiguration: Task-Skalierung

In dieser Konfiguration wurde die Threadanzahl auf eins festgelegt und die Taskanzahl erhöht. In den folgenden zwei Abbildungen werden die Ergebnisse der verschiedenen Pinning-Optionen gezeigt:



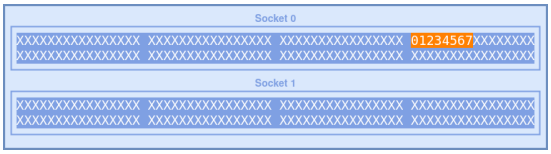
(a) Verschiedene Bindungsoptionen bei gesezter Standardverteilung



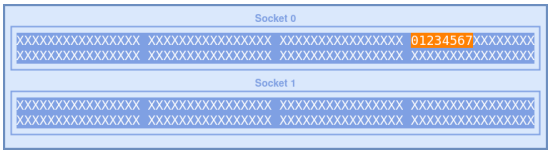
(b) Verschiedene Verteilungsoptionen bei gesetzter Standardbindung

Abbildung 5.18.: Durchschnittliche Laufzeit des HPL-Benchmarks für einen Thread und der Skalierung der Taskanzahl

In der Abbildung 5.18 ist ebenfalls zu sehen, dass alle Konfigurationen bis auf die Bindungsoption `rank` (Abbildung 5.18a) und die Verteilungsoption `--distribution=*:block:*` (Abbildung 5.18b), ähnliche Werte aufweisen. Die nachfolgenden Abbildungen zeigen das Pinning-Verhalten der zwei abweichenden Konfigurationen und der Bindungsoption `threads` mit Standardverteilung `--distribution=*:*:*` bei acht Tasks mit jeweils einem Thread.



(a) `rank`



(b) `--distribution=*:block:*`

(c) `threads`

Abbildung 5.19.: Pinning-Verhalten der unterschiedlichen Verteilungs- und Bindungsoptionen anhand von acht Tasks mit je einem Thread

Anhand der Grafiken in Abbildung 5.19 ist zu sehen, dass die Bindungsoption `rank` (Abbildung 5.19a) und die Verteilungsoption `--distribution=*:block:*` (Abbildung 5.19b) aufgrund der fehlenden Verteilung auf die verschiedenen NUMA-Domains eine längere Laufzeit benötigen. Zudem kommt es im gegebenen Beispiel zur Nutzung von zwei CCX-Einheiten bei der Bindungsoption `rank` (Abbildung 5.19a) und der Verteilungsoption `--distribution=*:block:*` (Abbildung 5.19b) bzw. zur Nutzung von acht CCX-Einheiten bei der Bindungsoption `threads` (Abbildung 5.19c). Das Nutzen von mehr CCX-Einheiten sorgt für mehr L3-Cache, der zur Verfügung steht, und kann somit eine bessere Leistung für diesen Benchmark ermöglichen.

5.2.4. Fazit des HPL-Benchmarks

Allgemein ist zu allen Grafiken zu erwähnen, dass die gemessene Zeit bei voller Auslastung eines Knotens für jegliche Konfigurationen nahezu identisch ist. Das lässt sich dadurch erklären, dass bei dieser Task- und Thread-Konfiguration alle Kerne des Knotens voll besetzt sind und das Pinning-Verhalten für die Auslastung keinen Unterschied macht. Jedoch ist dabei zu vermerken, dass eine Konfiguration, bei der die Threads einer Task auf viele NUMA-Domains verteilt werden, bei einer Anwendung mit vielen Gleitkommaoperationen wie dem HPL-Benchmark nicht zu empfehlen ist, da die Laufzeit durch Remote-Zugriffe deutlich verschlechtert wird. Außerdem ist zu erkennen, dass die hybride Konfiguration im Allgemeinen schnellere Messungen ergibt und das beste Ergebnis aller Konfigurationen bei 32 Tasks mit einem Thread, der Bindungsoption `threads` und der Verteilungsoption `--distribution=*:*:cyclic` mit 25,534 Sekunden gemessen wurde. Zudem zeigt sich dass die Standardbindungsoption `threads` mit der Standardverteilung `--distribution=*:*:*` in allen Varianten sehr gute Ergebnisse liefert.

6.Zusammenfassung und Ausblick

Nachfolgend wird das allgemeine Vorgehen dieser Arbeit und die zuvor dargestellten Ergebnisse des Leistungsvergleichs zusammengefasst. Abschließend wird ein Ausblick über die Thematik dieser Arbeit gegeben.

6.1. Zusammenfassung

Mithilfe des HPC Systems JUSUF des JSC wurden verschiedene Prozess-Pinning Strategien anhand der gemessenen Leistung von zwei standardmäßigen HPC-Benchmarks (STREAM- und HPL-Benchmark) analysiert. Das Prozess-Pinning beschreibt das Binden von Prozessen oder Threads an die Kerne eines Prozessors und kann beispielsweise durch das Vermeiden von Remote-Speicherzugriffen, der Ausnutzung von mehr Cache bzw. das Ausnutzen höherer Speicherbandbreiten, die Leistung einer Anwendung verbessern. Zur Konfiguration des Pinning-Verhaltens wurde der Workload Manager PsSlurm, der auf allen HPC Systemen des JSC zum Einsatz kommt, genutzt. Mithilfe des Workload Managers können Ressourcen und die zeitliche Ausführung mehrerer Prozesse gesteuert werden. Außerdem können verschiedene Bindungs- und Verteilungsoptionen für eine Anwendung eingestellt werden. Für den Vergleich der Benchmarks und dessen Leistungen wurden insgesamt sieben verschiedene Konfigurationen der Bindungs- und Verteilungsoptionen unter Variantion der Anzahl an Tasks und Threads getestet.

Für diese Arbeit wurde der STREAM-Benchmark, der die Speicherbandbreite für einfache Vektoroperationen misst, genutzt. Als weitere Metrik wurde die Laufzeit des HPL-Benchmarks, der ein lineares Gleichungssystem mithilfe der LU-Zerlegung löst, genutzt. Die Problemgrößen beider Benchmarks wurde an die Speichergröße des HPC Systems JUSUF angepasst.

Mithilfe der Programmumgebung JUBE war es möglich das Konfigurieren, Kompilieren und Ausführen der Benchmarks systematisch und automatisiert durchzuführen und die Ergebnisse zu analysieren. Dafür wurde für jede Konfiguration der Bindungs- und Verteilungsoptionen eine JUBE-Eingabedatei erstellt, die das Kompilieren und Konfigurieren der Task- und Threadanzahl und im Falle des STREAM-Benchmarks auch das Kompilieren übernahm. Zusätzlich übernahm JUBE das Ausführen beider Benchmarks.

Beim Analysieren des STREAM-Benchmarks wurde festgestellt, dass die Effizienz der physischen und virtuellen Kerne für diesen Benchmark keine deutlichen Unterschiede aufweist. Beim HPL-Benchmark, dessen Ergebnisse deutlich stärker von der Gleitkommaarithmetik abhängen, wurden jedoch deutliche Unterschiede bei der Leistung von physischen und virtuellen Kerne festgehalten. Entsprechend ist grundsätzlich die reine Nutzung physikalischer Kerne zu empfehlen, solange diese noch nicht vollständig belegt sind.

Außerdem wurde aus den Ergebnissen beider Benchmarks deutlich, dass die Verteilung von Tasks und Threads auf mehrere NUMA-Domains eine starke Leistungsverbesserung durch die gesteigerte Speicherbandbreite hervorbringt. Insbesondere kann beim HPL-Benchmark auch mehr L3-Cache durch Ausnutzung von mehr CCX-Einheiten zur Verfügung stehen und somit eine bessere Leistung erzielt werden. Dies ist jedoch nur

bei Anwendungen möglich, dessen Threads bzw. Tasks unabhängig voneinander arbeiten und somit nicht auf einen gemeinsamen Speicher angewiesen sind. Dies zeigte sich vorallem bei Thread-Konfigurationen des HPL-Benchmarks, bei der eine Verteilung der Threads einer Task über mehrere NUMA-Domains auch zu Leistungsverlusten führen konnte. Zudem lässt sich die Leistung nur zu einem gewissen Punkt steigern. Sobald mindestens ein Kern aller NUMA-Domains belegt wird, bleibt die Leistung für jeden folgenden Kern nahezu konstant.

6.2. Ausblick

Für die Analyse der Benchmarks wurde das HPC System JUSUF genutzt. Dies besitzt große Ähnlichkeit gegenüber der Architektur des HPC Systems JURECA und der Booster-Partition des HPC Systems JUWELS. Jedoch könnten die Benchmarks noch auf der Cluster-Partition von JUWELS getestet werden. Diese Partition weist eine andere Knotenstruktur, basierend auf Prozessoren von Intel, mit nur zwei NUMA-Domains auf. Daneben ließe sich die Auswirkung der gewählten Pinning-Konfigurationen auch auf die Grafikkarten-Anbindung in den verschiedenen Systemen übertragen, indem hier ein weiterer Benchmark gewählt wird, der die Einbindung von Grafikkarten ermöglicht.

Zusätzlich zu den verschiedenen Systemen und Partitionen könnte die Verteilung der Task und Threads, die sich in dieser Arbeit auf einen Knoten beschränkte, in fortführenden Analysen auf mehrere Knoten erweitert werden, um so auch eine Aussage zur Anbindung der Netzwerkkomponenten in den Knoten treffen zu können. Anhand dessen könnte zudem die Auswirkung der ersten Ebene der Verteilungsoption, die sich auf die Verteilung der Tasks auf die verschiedenen Knoten konzentriert, analysiert werden.

Abschließend kann noch darauf hingewiesen werden, dass sich die Problemgrößen der Benchmarks auf den Arbeitsspeicher gerichtet haben und somit den zur Verfügung stehenden L3-Cache vollständig ausgereizt haben. Es wäre somit auch möglich die Problemgrößen zu verkleinern, um diese an Messungen für den L3- bzw. auch L2 Cache-anzupassen.

A.Anhang

Acronyme und Fachbegriffe

JSC Jülich Supercomputing Centre

HPC High-Performance Computing

Scheduler Ein Scheduler verwaltet die zeitliche Ausführung mehrerer Prozesse in Betriebssystemen.

Ressourcenmanagers Ein Ressourcenmanager weist automatisiert Rechenressourcen zu.

Prozess Ein Prozess beschreibt ein in sich abgeschlossene Ausführungseinheit, die aus einem bzw. mehreren Threads besteht.

Task Siehe Prozess.

Thread Ein Thread ist ein Teil eines Prozesses

Verteilungsoption Die Verteilungsoptionen `--distribution` wird in SLURM die Verteilung von Task und Threads auf die verfügbaren Kerne bestimmt

Bindungsoption Mithilfe der Bindungsoptionen `--cpu-bind` können in SLURM die Task an bestimmte Kerne gebunden werden

NUMA Non-Uniform Memory Access

Cluster Ein Cluster beschreibt eine Anzahl von vernetzten Computern.

JUSUF HPC-System am JSC: Jülich Support for Fenix

JURECA HPC-System am JSC: Jülich Research on Exascale Cluster Architectures

JUWELS HPC-System am JSC: Jülich Wizard for European Leadership Science

CPU Central Processing Unit

Ressourcenmanagers Ein Ressourcenmanager weist automatisiert Rechenressourcen zu.

GPU Graphics Processing Unit

CCX Core-Complex

CCD Core-Complex-Die

Teraflops Ein Teraflops steht für 1 Billionen Rechenoperationen mit Gleitkommazahlen pro Sekunde.

SMP Symmetric Multiprocessing-Architekturen

Benchmarking Benchmarking beschreibt das Vergleichen von Ergebnissen oder Prozessen mit einem festgelegten Bezugswert.

MB/s MB/s ist eine angabe zur Geschwindigkeit von Datentransfer. Es steht für 'Megabytes per second' und beschreibt 1024 x 1024 Bytes/Sekunde.

Linpack Benchmark Die LINPACK-Benchmarks sind ein Maß für die Gleitkomma-Rechenleistung eines Systems.

MPI Message Passing Interface

BLAS Basic Linear Algebra Subprograms

Literaturverzeichnis

[1] *AMD Infinity Architecture*. https://www.amd.com/en/technologies/infinity-architecture. Stand 08.Juli 2021.

[2] Albino Aveleda et al. “Performance Comparison of Scientific Applications on Linux and Windows HPC Server Clusters”. In: vol. 29. Nov. 2010.

[3] *Batch system*. https://apps.fz-juelich.de/jsc/hps/juwels/batchsystem.html. Stand: 08. Juni 2021.

[4] Sergey Blagodurov et al. “A case for NUMA-aware contention management on multicore systems”. In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2010, pp. 557–558.

[5] Jalil Boukhobza et al. “Emerging NVM: A Survey on Architectural Integration and Research Challenges”. In: *ACM Transactions on Design Automation of Electronic Systems* 23 (Jan. 2018). DOI: 10.1145/3131848.

[6] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. “The LINPACK benchmark: past, present and future”. In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

[7] Stefan Gerlach. “Hochleistungsrechnen”. In: *Computerphysik*. Springer, 2019, pp. 89–101. ISBN: 978-3-662-59245-8.

[8] *JUBE 2.4.1 documentation*. https://apps.fz-juelich.de/jsc/jube/jube2/docu/. Stand: 22. Juni 2021.

[9] *JUBE Benchmarking Environment*. https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html. Stand: 22. Juni 2021.

[10] *JUSUF Cluster Partition Configuration*. https://apps.fz-juelich.de/jsc/hps/jusuf/cluster/configuration.html. Stand: 21. Juni 2021.

[11] Sebastian Lührs et al. “Flexible and Generic Workflow Management”. In: *Parallel Computing: On the Road to Exascale*. Vol. 27. Advances in parallel computing. International Conference on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015. Amsterdam: IOS Press, Sept. 1, 2015, pp. 431 –438. ISBN: 978-1-61499-620-0. DOI: 10.3233/978-1-61499-621-7-431. URL: https://juser.fz-juelich.de/record/808798.

[12] John D McCalpin. “Stream benchmark”. In: *Link: http://www.cs.virginia.edu/~mccalpin/papers/bandwidth* (1995).

[13] *Processor Affinity*. https://apps.fz-juelich.de/jsc/hps/jusuf/cluster/affinity.htmls. Stand: 22. Juni 2021.

[14] Julia Wellmann. “Entwicklung einer webbasierten Visualisierung des Prozess-Pinnings auf HPC-Systemen”. In: Course work, FH Aachen, 2020. URL: https://juser.fz-juelich.de/record/893380.