Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Analyse, Entwurf und Implementierung einer Zuordnungsstrategie für OpenMP-Prozesse in planungsbasierten Cluster-Verwaltungssystemen

Henning Erdweg
Matrikelnummer: 5476026
henning.erdweg@fu-berlin.de

Betreuer: Barry Linnert
Eingereicht bei: Barry Linnert
Zweitgutachter: Prof. Dr.-Ing. Jochen Schiller

Berlin, 11. August 2023

Inhaltsverzeichnis

1 Einleitung

Die von Foster und Kesselmann [4] skizzierte Idee eines Grids, beschreibt ein verteiltes System, in dem Ressourcen geteilt werden. Die Idee beschränkt sich hierbei nicht auf physische Ressourcen, sondern erstreckt sich allgemein auf teilbare Ressourcen, sodass zum Beispiel auch Informationen umfasst sind. Vorhandene Ressourcen werden hierbei zur Verfügung gestellt und im Anschluss Anwendern transparenter Zugriff gewährt.

In Bezugnahme auf High Performance Computing (HPC) können verschiedene Clustersysteme über ein Grid geteilt werden (siehe Abbildung 1). Die Idee eines Grids ermöglicht den Cluster-Verwaltungssystemen ihre Ressourcen nach außen zur Verfügung zu stellen.

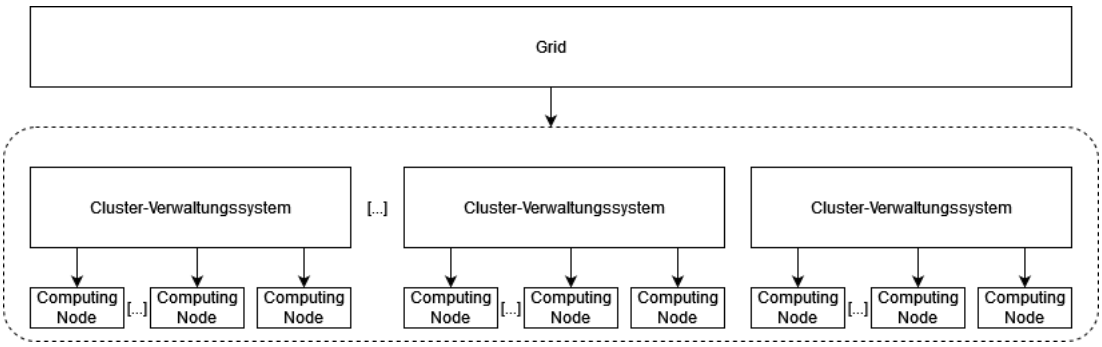

Abbildung 1: Hierarchische Darstellung des Zusammenhangs zwischen Grid und Cluster.

Die vorliegende Arbeit beschäftigt sich mit dem Einsatz von Open Multi-Processing (OpenMP) in planungsbasierten Cluster Verwaltungssystemen. Der planungsbasierte Scheduler ist Gegenstand aktueller Forschung und soll eine verbesserte Verwaltung einzelner Cluster erzielen.

Zunächst beschäftigt sich diese Arbeit im Allgemeinen mit HPC-Systemen und OpenMP und diskutiert anschließend aufkommende Problemstellungen bezüglich des Einsatzes von OpenMP im Kontext eines planungsbasierten Schedulers in HPC-Systemen.

2 High Performance Computing

Als High Performance Computing Systeme (HPC-Systeme) werden heute zumeist parallele Clustersysteme[1] bezeichnet, die aus beliebig vielen zusammengeschalteten Rechnern bestehen und somit ein sehr leistungsfähiges Gesamtsystem schaffen. Die Leistungsfähigkeit der Rechnerverbunde geht dabei weit über Heimcomputer hinaus. Diese werden für spezielle Probleme eingesetzt, die eine immense Rechenleistung benötigen. Die Zusammenschaltung von Computern kann dabei homogen, also mit den gleichen Computerkomponenten wie Speicher und Prozessor erfolgen oder aber auch heterogen, wobei Computer mit verschiedensten Computerkomponenten miteinan-

[1] Die vorliegende Arbeit betrachtet nur Clustersysteme im HPC-Bereich.

der in einen Verbund geschaltet werden. Die zusammengeschalteten Rechner werden im Kontext von HPC-Systemen als Computing Nodes bezeichnet und laufen in aller Regel mit Betriebssystemen aus der Linux-Familie. Von den derzeit leistungsfähigsten fünfhundert HPC-Systemen laufen alle Clustersysteme mit Betriebssystemen der Linux-Familie. [24]

Der aktuell weltweit leistungsstärkste Supercomputer ist der Hewlett Packard Enterprise Frontier (OLCF-5) des Oak Ridge Leadership Computing Facility in den USA mit 8.699.904 Prozessorkernen. Jeder Computing Node läuft mit einer AMD Epyc 7453s „Trento“ CPU, welche wiederum 64 Kerne beinhaltet. [24]

Der eigentliche Nutzen eines HPC-Systems kann erst dann zum Vorschein treten, wenn es Algorithmen gibt, die parallel auf den verschiedenen Computing Nodes verteilt werden können bzw. auch auf einem einzigen Knoten parallel ausgeführt werden können. Da Heimcomputer heutzutage standardmäßig mehrkernige Prozessoren haben, die eine physische parallele Ausführung erlauben, ist das Problem eines parallelen Algorithmus hier ebenfalls relevant zu untersuchen. Nach Amdahls Gesetz wird der Geschwindigkeitszuwachs jedoch durch den sequenziellen Teil eines jeden Programms beschränkt, da dieser nicht parallel ausgeführt werden kann. Daraus wird deutlich, dass es nicht nur entscheidend ist leistungsfähige HPC-Systeme zu designen, sondern auch die entsprechenden Programme. Hierbei müssen schon vor der Implementierung Entwurfsentscheidungen getroffen werden, die eine möglichst geringe Abhängigkeit zwischen Prozessen und Threads von parallelen Programmen beinhalten. Der Kommunikationsaufwand und die Datenabhängigkeit sollten daher schon während der Konzeption beachtet und möglichst so organisiert werden, dass sie nur einen geringen Einfluss auf den Programmfluss haben. Ein bekanntes Entwurfsmuster zur Konzeption solcher Algorithmen wird von Fosters Design Methode beschrieben. [1]

Im Optimalfall ist die Auslastung eines HPC-Systems dann so hoch, dass es kaum Leerlaufzeiten gibt, in denen Prozessoren von Computing Nodes im Leerlauf sind. Hierbei bedarf es aber auch einer organisationsinternen Planung, da HPC-Systeme zum Teil nur einen sehr exklusiven Zugriff gestatten, um stets die aktuell wichtigsten Programme ausführen zu können. Das widerspricht natürlich grundsätzlich der Idee eines HPC-Systems, da die Ressourcenintensität inklusive der Anschaffungs- und Betriebskosten sehr hoch ist und eine zu geringe Auslastung stark die Wirtschaftlichkeit des Systems infrage stellt. Die in „Plan Based Thread Scheduling on HPC-Nodes“ [5] vorgeschlagene Methode eines planungsbasierten Schedulers liefert hierfür einen neuen Ansatz, welcher Garantien für die Laufzeit eines Programms angibt und somit auch die gemeinsame Nutzung eines HPC-Systems in Bereichen interessanter macht, in denen eine garantierte Verfügbarkeit des Systems gewährleistet werden muss. Dies könnte zum Beispiel bei einer Wettersimulation der Fall sein, da das Ergebnis der Simulation zeitkritisch für einen aktuellen Wetterbericht ist.

2.1 Einsatzgebiete

HPC-Systeme kommen in den verschiedensten Problembereichen zum Einsatz. So werden zum Beispiel Genomsequenzierungen, maschinelles Lernen oder komplexe

Simulationen in der Autoindustrie mit Hilfe von HPC-Systemen durchgeführt. Die folgenden zwei Beispiele der Wettersimulation und der Einsatz in Forschung und Wissenschaft sollen einen genaueren Blick in Einsatzgebiete geben. Zudem zeigen die aufgeführten Beispiele, dass immer mehr Branchen HPC-Systeme benötigen.

2.1.1 Wettersimulation

Eines der ältesten Anwendungsgebiete ist die Wettersimulation. Schon ab den 1950er Jahren wurden die ersten computergenerierten Vorhersagen der Atmosphäre vorgenommen. Heutzutage ist der Einsatz von HPC-Systemen in der Wettervorhersage obligatorisch und wird von den Wetterdiensten wie dem European Center for Medium Range Weather Forecasts (ECMWF), dem U.S. National Weather Service (NOAA) oder dem deutschen Wetterdienst (DWD) verwendet. [20]

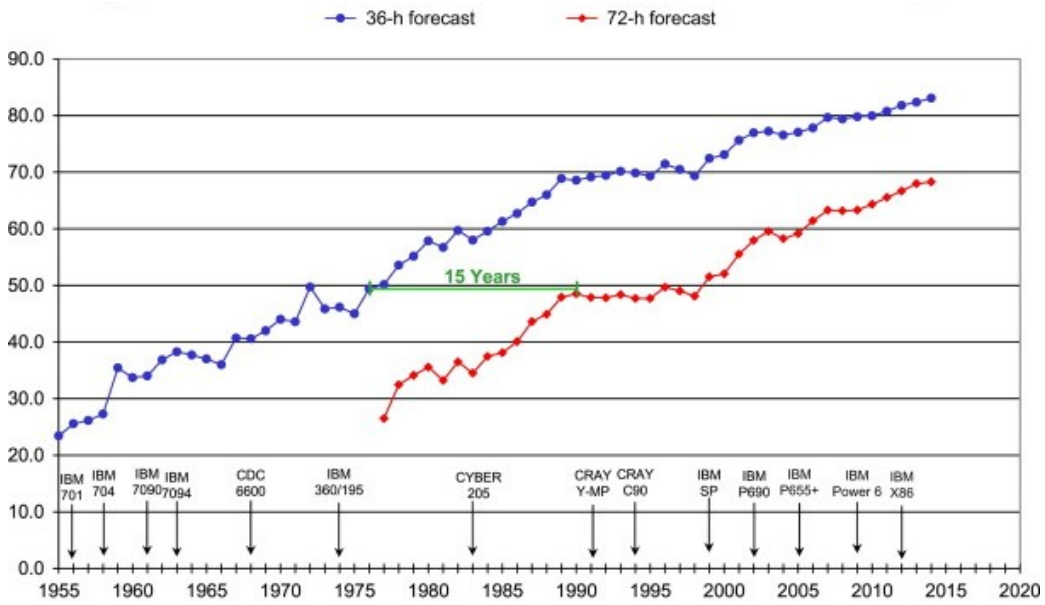

Abbildung 2: Korrelation der Vorhersagegenauigkeit (100 = perfekt) mit der Rechenleistung beim NOAA: Die Vorhersagegenauigkeit nimmt linear zu, wenn die Rechenleistung exponentiell bei aufeinanderfolgenden Generationen von HPC-Systemen steigt. [20]

Die in Abbildung 2 dargestellte Auswertung der Vorhersagegenauigkeit aus dem Zeitraum von 1955 bis 2015 des NOAA legt dar, das bei einer exponentiellen Entwicklung der Rechenleistung der eingesetzten HPC-Systeme eine lineare Verbesserung der Vorhersagegenauigkeit eintritt. Die zukünftige Herausforderung der Wetterinstitute für die Wettervorhersage wird darin bestehen, in Zeiten von Exascale[2] HPC-Systemen wie dem Frontier weiterhin lineare Verbesserungen der Vorhersagegenauigkeit zu erzielen. [20]

[2] 10^{18} IEEE 754 Double Precision (64-bit) Operationen (Multiplikationen und/oder Additionen) pro Sekunde (exaFLOPS).

2.1.2 ITER Fusionsreaktor

Ein neues Forschungsgebiet im Bereich der HPC-Systeme wird von Forschern des Oak Ridge National Laboratory erkundet. Durch die Simulation eines Tokamak Plasma-Fusionsreaktors sollen neue Erkenntnisse für den ITER Tokamak Reaktor in Frankreich gewonnen werden. Dabei ist es für die Laufzeit der Simulation entscheidend, auf einem Exascale System betrieben zu werden, da im besten Fall ein Geschwindigkeitszuwachs um den Faktor 50 erzielt werden kann. [19, 26]

2.2 Scheduling

Zu den Kernaufgaben eines jeden Betriebssystemkernels gehört es, Prozesse zu verwalten und zu entscheiden, welcher Prozess wie lange auf einem Prozessorkern ausgeführt werden soll. In HPC-Systemen ist daher eine der Hauptaufgaben ein optimales Scheduling zu implementieren, welches ebenfalls durch den Einsatz des planungsbasierten Scheduler angestrebt wird.

Dabei gibt es einige grundlegende Überlegungen, nach welchen Kriterien Prozesse ausgeführt werden sollen. Ein Hauptziel ist es möglichst viele Prozesse pro Zeiteinheit abzuarbeiten, also einen hohen Durchsatz (Throughput) zu erzielen. Zudem sollte die Antwortzeit (Response time) von Prozessen möglichst gering sein, sodass die Interaktivität von Prozessen gewährleistet ist und zudem eine gewisse Fairness hergestellt werden kann. Zuletzt wird eine hohe Prozessoreffizienz (Processor efficiency) angestrebt, welche die Auslastung des Prozessors als nutzbare Ressource maximieren soll. Alle Ziele gleichzeitig zu optimieren ist jedoch schwierig, da ein maximaler Durchsatz zu einer unvorhersagbaren Antwortzeit führt und eine kurze Antwortzeit zu einem verringerten Durchsatz. [25]

Im Spannungsfeld dieser Ziele wird deutlich, dass zwischen einem Scheduling für interaktive Benutzerbetriebssysteme und einem für systemorientierte Betriebssysteme unterschieden werden muss. Interaktive Benutzerbetriebssysteme wollen vor allem die Antwortzeit (Response time) minimieren, sodass möglichst viele Prozesse ausgeführt werden können, die nicht verhungern (Starvation). Systemorientierte Betriebssysteme wollen vor allem den Durchsatz und die Prozessorauslastung maximieren. Dabei wird eher der Fokus auf die Hardware gelegt, um diese aus einer Ressourcensicht optimal auslasten zu können. [25]

2.2.1 Linuxscheduler

Im Linuxkernel kommen verschiedene Scheduling Strategien zum Einsatz. Neben der Earliest Deadline First (EDF) Strategie, der First-In First-Out (FIFO) Strategie und der Round-Robin Strategie soll an dieser Stelle vor allem der Completely Fair Scheduler (CFS) erwähnt sein. Dieser wird als Default Scheduler verwendet und verfolgt die klassischen Ziele eines interaktiven Benutzerbetriebssystemschedulers und hat den Anspruch so fair wie möglich zu sein. [2]

2.2.2 Warteschlangen-basiertes Scheduling in HPC-Systemen

In aktuellen HPC-Systemen kommt ein Warteschlangen-basiertes Scheduling zum Einsatz, welches am ehesten die Ziele eines systemorientierten Betriebssystems verfolgt. Ein einfacher Ansatz ist die Implementierung eines FIFO Scheduling, welches die Jobs (Programm und Eingabedaten) nach der Reihenfolge abarbeitet, in der sie eingetroffen sind. Ein Warteschlangen-basiertes Scheduling hat den großen Vorteil, dass es einfach umzusetzen ist, da die Zuteilung ohne weiteres Wissen erfolgen kann. Die Ressourcenvergabe kann dabei mitunter aber nicht optimal sein, sodass einzelne Computing Nodes nicht ausgelastet sind und somit die grundsätzlichen Ziele des Scheduling verfehlt werden. Um das Problem zu lösen, kann dieser Ansatz um die Methode des Backfilling erweitert werden, wobei der Scheduler die statische Reihenfolge der Warteschlange aufweicht und spätere Jobs parallel mit dem aktuellen Job ausführt. Hierbei werden jedoch genauere Kenntnisse über das HPC-System benötigt, um abschätzen zu können, ob ein weiterer Job gestartet werden kann oder nicht. [8]

2.2.3 Planungsbasierter Scheduler in HPC-Systemen

Der für diese Arbeit maßgeblich zu betrachtende planungsbasierte Scheduler ist Gegenstand aktueller Forschung und führt die Threads nach einem vordefinierten Plan aus, der vor der Ausführung des HPC-Jobs komplett geplant wird. Der Scheduler siedelt sich im Konzept von Grid-Systemen in der Middleware an (Cluster-Verwaltungssysteme) und implementiert hier das neue Konzept eines Virtual Resource Manager (VRM) mit der Definition von Quality of Services (QoS). Der Plan setzt sich aus einem Task Interaction Graph (TIG) und einem Task Precedence Graph (TPG) zusammen. Bei beiden direkten Graphen stellen die Knoten Tasks dar, wobei die Kanten von einem Task t_i auf den nächsten Task t_j zeigen. Die Kanten des TPG geben an, dass t_i vor t_j ausgeführt werden muss. Die Kanten des TIG geben wiederum an, dass ein Kommunikationsfluss von t_i nach t_j existiert. Der verwendete Plan setzt sich aus dem TIG und TPG zusammen, da sowohl die Reihenfolge des Scheduling bekannt sein muss als auch mögliche Kommunikationsressourcen vorbehalten werden müssen. Jeder Task besitzt eine ID auf welchem Computing Node dieser ausgeführt werden soll, eine Thread-ID und eine Start- und eine Endfrist, welche einen Rahmen für die spätesten möglichen Ausführungszeiträume geben. Der Graph des Planes eines HPC-Systems wird in Teilpläne (genannt Scheduling Pläne) für die Computing Nodes aufgeteilt, um die Last auf die verschiedenen Knoten zu verteilen. [5]

Der Hauptunterschied des planungsbasierten Schedulers im Vergleich zu herkömmlichen Schedulern ist, dass die gesamten Scheduling Pläne, sowohl der Gesamtplan als auch die Teilpläne, vorberechnet sind. Das ermöglicht erst die Einführung von Service Level Agreements (SLA) als Spezifizierung einer QoS, da der Scheduler lediglich einer einfachen Entscheidungslogik folgen muss, welcher Thread als nächstes für wie lange ausgeführt werden soll. Ein SLA ist der ausgehandelte Vertrag zwischen dem ressourcennutzenden Client und dem VRM. Im Verhandlungsprozess wird bereits ein Plan ausgehend von Vorhersagemodellen und historischen Daten erstellt, um die Laufzeit zu bestimmen. Eine ausgehandelte SLA beruft sich dann auf diesen Plan zum

Scheduling des Systems. Ebenfalls ist es möglich Strafbeträge bei Nichteinhaltung des Vertrages festzulegen.

Sollten neue Jobs zur Ausführung bereitstehen, kann der VRM zur Laufzeit den Plan anpassen und somit dynamisch auf neue Anforderungen reagieren. Dies kann zudem der Fall bei Vorhersagefehlern sein, sodass eine Anpassung des Plans nötig ist. [5]

3 Open Multi-Processing

Open Multi-Processing (OpenMP) ist ein Application Interface (API) für multi-threaded shared-memory Programme. OpenMP wird für C, C++ und Fortran angeboten, wobei sowohl verschiedene Instruction set architectures (ISA) als auch Betriebssysteme unterstützt werden. Das im Kontext von HPC-Systemen oft verwendete ISA x86-64 und das Betriebssystem Linux werden auch unterstützt. Es existieren verschiedene Implementierungen von der OpenMP API. Die Open-Source-Implementierung libgomp ist in der GNU Compiler Collection (GCC) enthalten. Das Flag *-fopenmp* muss bei der Kompilation gesetzt werden, um die OpenMP-Parallelisierung zu verwenden. Weitere Compiler mit OpenMP-Support bieten unter anderem aber auch Intel, AMD, IBM, Microsoft und Oracle an. Allgemein bietet es sich bei der Konzeption von HPC-Systemen zur Kommunikation zwischen Computing Nodes an, auf MPI zu setzen und für die Parallelisierung auf einem Computing Node mit OpenMP zu arbeiten. OpenMP operiert nach dem Fork-Join-Prinzip und erstellt ausgehend von einem Initial Thread weitere Worker Threads, welche innerhalb einer Parallel Region einen Codeabschnitt parallelisieren (siehe Abbildung 3). Der erzeugende Thread einer Parallel Region[3] erstellt zur Bearbeitung der Parallel Region ein Team, welches aus dem erzeugenden Thread, genannt Primary Thread (Thread-Id 0) und mindestens einem Worker Thread (Thread-Ids 1 bis Anzahl von Threads im Team - 1) besteht. Der Initial Thread selbst gehört zu einer Implicit Parallel Region, sodass der sequenzielle Codeanteil intern ebenfalls als Task repräsentiert wird (siehe Kapitel 3.2). [21, 23]

Der Einsatz von OpenMP im Gegensatz zu der Verwendung von *pthreads* schafft einige Vorteile. Der Sourcecode ist in der Zeilenanzahl deutlich reduziert, da die komplette Komplexität der Threaderstellung und Verwaltung innerhalb weniger Zeilen wegabstrahiert wird. Dadurch ist es auch einfacher ein paralleles Programm zu schreiben, da die Threadverwaltung von OpenMP übernommen wird. Die geringere Komplexität sollte sich auch in den Entwicklungszeiten widerspiegeln und die Wartbarkeit des Codes ebenfalls deutlich vereinfachen. Programmierfehler sind mit OpenMP auch einfacher zu vermeiden und sollten im Resultat zuverlässigere Programme erzeugen.

Zusätzlich zeichnet sich OpenMP auch noch durch eine hohe Portierbarkeit aus, wodurch eine hohe Wiederverwendbarkeit des Codes erreicht werden kann und die Adaption für neue Plattformen weniger zeitaufwendig ist.

[3] Außerhalb von Nested-Konstrukten ist der erzeugende Thread immer der Initial Thread.

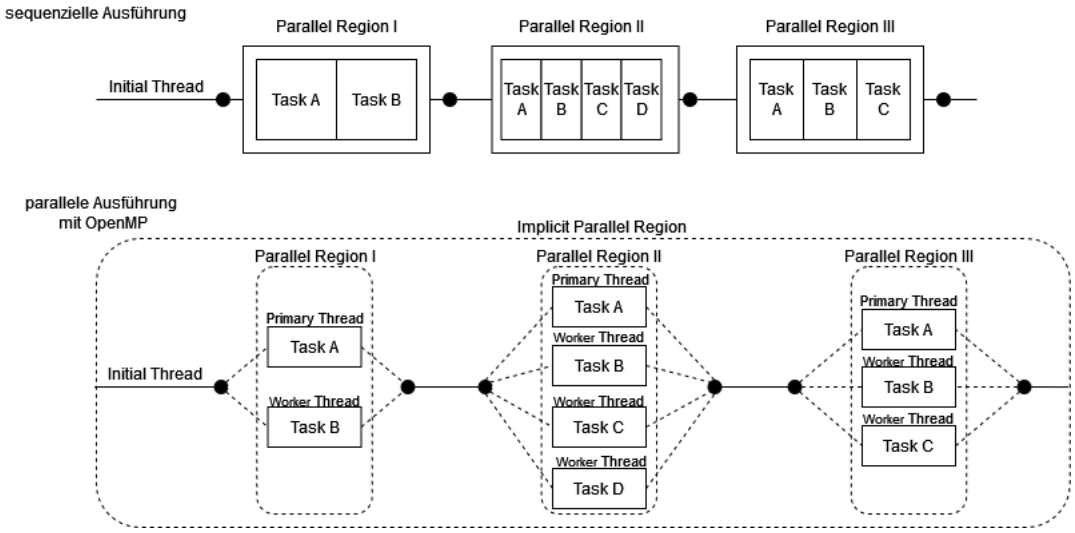

Abbildung 3: Fork-Join-Modell in schematischer Darstellung, wobei der sequenzielle Ausführungsplan direkt mit dem parallelen Ausführungsplan unter OpenMP korreliert, mit dem Unterschied, dass die Tasks mit OpenMP parallel bearbeitet werden.

3.1 Direktiven

OpenMP-Direktiven bilden die Kernstruktur eines jeden OpenMP-Programms und starten immer mit der Phrase *#pragma omp* (siehe Abbildung 4). Dabei dient die *#pragma*-Phrase als Kennwort für den Präprozessor, welcher den Code in tatsächlich parallelisierten Code verwandelt. Danach kommt ein spezifisches Kennwort für unterschiedliche Parallelisierungsmechanismen und optionale Klauseln. Insofern eine sequenzielle Ausführung ohne OpenMP gewünscht ist, kann der Code auch übersetzt werden, indem die Direktiven ignoriert werden und somit das Programm sequenziell ausgeführt wird. Die nachfolgend betrachteten Direktiven geben nur einen Überblick über die wichtigsten Direktiven, stellen aber nicht das vollständige Bild aller Parallelisierungsmöglichkeiten mit OpenMP dar. [21]

```
# pragma omp directive-name [ clause [ clause ...]]
{
  // code scope of parallel executed tasks
}
```

Abbildung 4: Allgemeiner Syntaxaufbau aller OpenMP-Befehle.

3.1.1 Parallel Regions

Die einfachste Möglichkeit Code zu parallelisieren ist die Verwendung von *#pragma omp parallel* (siehe Abbildung 5). Dabei kann als optionale Klausel *num_threads(n)* angegeben werden, wobei n für die Anzahl zu verwendender Threads steht. Es ist zu beachten, dass bei der Verwendung von Parallel Regions alle Threads denselben Code ausführen. [21]

```
#pragma omp parallel
{
  // code scope of parallel executed implicit tasks
}
```

Abbildung 5: Syntax einer Parallel Region ohne Angabe von optionalen Klauseln.

3.1.2 Workshare For-Schleifen

Innerhalb einer Parallel Region kann die Arbeit eines Threads weiter unterteilt werden, indem das *#pragma omp for* verwendet wird (siehe Abbildung 6). Dabei werden die Iterationen der For-Schleife auf die verschiedenen Threads aufgeteilt. [21]

Die Aufteilung kann dabei nach verschiedenen Schedulingmechanismen erfolgen. [21]

Bei **statischem** Scheduling werden die Iterationen in Chunks der Größe *chunk_size* unterteilt und mit dem Round-Robin Verfahren verteilt. Jeder Chunk beinhaltet *chunk_size* Iterationen, mit Ausnahme der letzten Iteration, welche auch weniger Iterationen haben kann. Die Reihenfolge der Chunkvergabe des Round-Robin Verfahrens erfolgt in sortierter Reihenfolge nach der Thread-Id. Der Primary Thread bearbeitet also immer den ersten Chunk und der erste Worker Thread den zweiten Chunk usw. [21]

Bei **dynamischem** Scheduling führt jeder Thread zunächst einen Chunk aus und fragt nach Beendigung der Ausführung den Nächsten an. Die Verteilung der Chunks erfolgt gleichmäßig, mit Ausnahme des letzten Chunks, welcher kleiner sein kann. Ist *chunk_size* nicht definiert wird default mäßig 1 verwendet. [21]

Guided Scheduling verhält sich ähnlich wie dynamisches Scheduling, jedoch nimmt die *chunk_size* mit fortschreitender Iterationsdauer ab. [21]

Zuletzt seien die Optionen **auto** und **runtime** erwähnt. **Auto** entscheidet zur Laufzeit, welcher Scheduler verwendet wird und **runtime** liest die Information aus der *run-sched-vartext* Umgebungsvariable, welche vor Programmausführung gesetzt werden kann. Explizite Angaben eines Scheduling im Programmcode überschreiben diese jedoch. [21]

```
#pragma omp parallel
{
  #pragma omp for  (kind [ , chunk size ])
    for (int i = 0; i < 4; i++)
    {
      // code scope of parallel executed implicit tasks
    }
}
```

Abbildung 6: Syntax einer Workshare For-Schleife.

3.1.3 Workshare Sections

Zur Parallelisierung von unabhängigem verschiedenartigem Code bietet sich die Verwendung von Sections an (siehe Abbildung 7). Diese verwenden intern ebenfalls Workshare-Konstrukte, die unter den teilnehmenden Threads aufgeteilt werden. Dabei wird analog zu dem dynamischen Scheduling für For-Schleifen ein Scheduling verwendet, welches den Threads jeweils eine Section initial zuteilt und bei Fertigstellung die nächste zu bearbeitende Section vergibt. Die Reihenfolge der Vergabe erfolgt dabei chronologisch und einem Thread wird nur eine Section zur selben Zeit zugeteilt. Die Sectionreihenfolge ergibt sich aus der Reihenfolge des Programmcodes. [21]

```
#pragma omp parallel sections
{
  #pragma omp section
    // code scope of parallel executed implicit task

  #pragma omp section
    // code scope of parallel executed implicit task
}
```

Abbildung 7: Syntax eines Workshare Section-Konstrukts.

3.2 Threads vs. Tasks

Im Kontext von OpenMP wird ebenfalls zwischen Threads und Tasks unterschieden. Ein Thread ist eine Ausführungseinheit mit Stack und assoziiertem *threadprivate* [4] Speicher. Dabei werden OpenMP-Threads als die Threads spezifiziert, die von der OpenMP-Implementation verwaltet werden. [21]

```
    #pragma omp task
    // code scope of parallel executed explicit task
```

Abbildung 8: Syntax eines Explicit Task.

Tasks wiederum spezifizieren ausführbaren Code und die dazugehörigen Daten. Tasks werden von der OpenMP-Implementierung gescheduled und anschließend von den Threads ausgeführt. Ohne nähere Spezifikation eines Tasks spricht man daher auch von Implicit Tasks, die durch ein paralleles Konstrukt erzeugt werden. Explicit Tasks (siehe Abbildung 8) können von allen Threads des Teams der Parallel Region ausgeführt werden. Standardmäßig sind Tasks *tied*, d.h. sie können nur von einem Thread ausgeführt werden, nachdem die Ausführung begonnen hat. Das strikte Fork-Join-Prinzip entfällt bei der Verwendung von Tasks. Wenn ein Thread einen Task erstellt, führt er diesen direkt aus oder platziert ihn in einem Taskpool, sodass dieser später von den Threads aus dem Team ausgeführt werden kann. Die Synchronisation kann explizit mit dem Befehl *#pragma omp taskwait* erfolgen, wobei auch auf alle Child-Tasks

[4] Die *threadprivate* Direktive legt fest, dass die Variablen geklont werden, wobei jeder Thread seine eigene Kopie hat.

gewartet wird. Der Befehl *#pragma omp barrier* kann ebenfalls zur Synchronisation verwendet werden. [21]

4 Untersuchungsgegenstand

Im Kontext von planungsbasierten HPC-Systemen ist es unerlässlich, möglichst viele Informationen über die geplanten Aufgaben zu erhalten bzw. genauestens einschätzen zu können, wann welche Prozesse involviert sind, wie ihre Laufzeiten sind und welche Ressourcen diese optimalerweise benötigen. Die optimalen Ressourcen können sich zum Beispiel aus der Anzahl der Ressourcen aber auch der Art der Ressourcen ergeben. So ist es denkbar, dass HPC-Knotenressourcen unterschiedlich sind und deshalb unterschiedliche Aufgaben präferieren.

Daher sind die gängigen eingesetzten Werkzeuge in HPC-Systemen von besonderer Bedeutung. OpenMP wird neben OpenMPI und pthreads traditionell im Kontext von HPC-Systemen genutzt, wodurch der grundsätzliche Bedarf eines Einsatzes auch im planungsbasierten Kontext gegeben ist. Zusätzlich spielt der gleichzeitige Einsatz von sowohl OpenMPI und OpenMP eine immer größere Rolle, da damit programmierte Algorithmen gute Laufzeitergebnisse hervorbringen. So hat der ECMWF z.B. festgestellt, dass ihre Wettersimulationen am performantesten auf ihrem HPC-System laufen, wenn pro MPI Task 8 OpenMP-Threads verwendet werden bei insgesamt 48 Computing Nodes. [7, 9]

Hauptuntersuchungsgegenstand ist daher, wie Threads eines Computing Nodes mit OpenMP erzeugt werden und Prozessorkernen zugeordnet werden. Die Zuordnung der Threads zu Kernen ist essenziell für den planungsbasierten Scheduler, da sichergestellt werden muss, dass der vordefinierte Ausführungsplan auch zur Laufzeit eingehalten wird. Hierfür ist es nötig festlegen zu können, dass ein bestimmter Thread auch tatsächlich auf dem gewünschten Kern ausgeführt wird. Dazu müssen vor allem die OpenMP Direktiven untersucht werden und wie diese vom Compiler übersetzt werden und sich dadurch auf den Programmablauf auswirken.

Zudem soll erörtert werden, wie mit OpenMP ein Pinning und eine Verwaltung aller OpenMP-Threads zu Prozessorkernen erfolgen kann. Dazu sollen die verfügbaren Möglichkeiten mit der Benutzung von OpenMP untersucht werden. Besonders soll evaluiert, wie die erzeugten Threads aktiv einem Kern zugeordnet werden können, sodass aus den ermittelten Informationen über einen OpenMP-Prozess eine sinnvolle Ressourcenzuteilung erfolgen kann.

Zuletzt soll ein Prototyp entwickelt werden, welcher die gewonnenen Erkenntnisse aus den ersten beiden Schritten verwendet und in den Kontext eines planungsbasierten Schedulers integriert. Dieser soll in der Lage sein mit der Grid Middleware des HPC-Systems zu kommunizieren, um eine ideale Belegung der Prozessorressourcen auf dem Knoten bestimmen zu können.

5 Analyse

Zur Evaluation des Einsatzes von OpenMP im Kontext von planungsbasierten Cluster-Verwaltungssystemen fokussiert sich diese Arbeit auf die Open-Source-Implementierung GNU libgomp von OpenMP sowie auf Linux Betriebssysteme, die in allen Top 500 Cluster-Verwaltungssystemen zum Einsatz kommen.

Im Folgenden wird OpenMP als Synonym für die spezifische Implementierung von GNU libgomp verwendet.

5.1 Native POSIX Thread Library

Per Default verwendet OpenMP die Native POSIX Thread Library (NPTL), welche für Linux einen POSIX Threads (pthreads) kompatiblen Standard anbietet. Die hier untersuchte Implementierung bezieht sich auf die NPTL-Implementierung aus der GNU C Library (glibc). NPTL verwendet zur Absicherung die fast user-space mutexes (futex) [11] Implementierung des Kernels. Es wird ein 1:1 Threading Modell umgesetzt, welches jedem neuen Thread genau einen neuen Kernel Thread zuordnet. Zunächst erstellt OpenMP ein neues Team, welches die parallelen Anweisungen ausführen soll. Dafür erzeugt es nach der Initialisierung in GOMP_parallel_sections() neue Threads in gomp_team_start(). Dazu wird intern ein Threadpool verwendet, der neue Threads nur nach Bedarf anlegt und somit den Overhead einer späteren Threaderstellung minimieren soll. Überschüssige Worker Threads außerhalb einer Parallel Region befinden sich in einer Idle-Schleife, der statischen *pthread_create* Startfunktion *gomp_thread_start()*[5]. [3, 12]

```
/* Backtrace eines Beispiel OpenMP-Programms mit gdb nach der
   Verwendung von #pragma omp parallel sections. Die verwendeten
   Bibliotheken sind libgomp aus gcc 13 und glibc 2.37
*/
#0  clone3 () at clone3.S:42
#1  __clone_internal () at clone-internal.c:54
#2  create_thread () at pthread_create.c:297
#3  __pthread_create_2_1 () at pthread_create.c:833
#4  gomp_team_start () at team.c:858
#5  GOMP_parallel_sections () at sections.c:237
#6  main () at example_program.c:34
```

Abbildung 9: Bereinigter Backtrace der Threaderstellung eines OpenMP-Programms (vollständig in Abbildung 21).

Mittels der POSIX-konformen Implementierung von pthreads in glibc wird nun der Syscall *clone3* verwendet, welcher ähnlich wie ein Prozessfork funktioniert und einen neuen light weight process (LWP) im Kernel erstellt. LWP sind die Linux-Interpretation von Threads und unterscheiden sich nicht maßgeblich zum Aufbau von Prozessen.

[5] Die statische Funktion *gomp_thread_start()* wird immer beim Aufruf von *pthread_create* verwendet und dient als Wrapper-Funktion für die eigentlich auszuführende Funktion.

Beide werden über die Struktur *task_struct* verwaltet und erhalten eine einzigartige PID [6]. Zudem stellt jeder *task_struct* einen unabhängigen Ausführungskontext dar. [2]

Die folgenden clone Flags werden von NPTL verwendet, um eine POSIX-konforme Thread-Implementierung für Linux bereitstellen zu können. Der Syscall kann allgemein als die generische Version der Fork-Funktion gesehen werden. Prozessweit müssen laut POSIX-Definition unter anderem die Prozess-ID, Vaterprozess-ID, Prozessgruppen-ID, Session-ID, Dateideskriptor-ID und das aktuelle Arbeitsverzeichnis geteilt werden. Pro Thread müssen unter anderem die Thread-ID, Signal Mask, Scheduling Policies und CPU Affinität vergeben werden. [15]

```
// glibc/nptl/pthread_create.c: Zeile 279
  const int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES
  | CLONE_SYSVSEM | CLONE_SIGHAND | CLONE_THREAD
  | CLONE_SETTLS | CLONE_PARENT_SETTID
  | CLONE_CHILD_CLEARTID
  | 0);
```

Abbildung 10: Verwendete Flags für den Syscall clone3() der NPTL-Implementierung zur Erzeugung eines LWP.

Die wichtigsten Flags zur Erstellung eines LWP sind im Folgenden ausführlicher erklärt:

- CLONE_VM definiert, dass der aufrufende Thread und der erzeugte Thread in demselben virtuellen Adressraum ausgeführt werden. Durchgeführte Schreibzugriffe auf den Speicher sind für beide Threads sichtbar. [10]
- CLONE_FS definiert, dass sich der aufrufende Thread und der erzeugte Thread die gleichen Dateisysteminformationen teilen. Dies beinhaltet die Wurzel des Dateisystems, das aktuelle Arbeitsverzeichnis und die umask. [10]
- CLONE_FILES definiert, dass sich der aufrufende Thread und der erzeugte Thread dieselbe Dateihandletabelle teilen. Alle Handles sind sowohl im aufrufenden Thread als auch im erzeugten Thread gültig. [10]
- CLONE_THREAD definiert, dass der aufrufende Thread in derselben Thread-Gruppe wie der erzeugte Thread platziert wird. Thread-Gruppen stellen eine Gruppe von Threads dar, die sich eine einzige PID teilen. Intern ist diese gemeinsame PID der sogenannte Thread-Gruppen-Identifikator (TGID) für die Thread-Gruppe. Die eigentliche Rückgabe des getpid() Syscalls ist daher auch die TGID. Die systemweite TID kann mit der Funktion gettid() ermittelt werden. [10]

Die Umsetzung in OpenMP verwendet also standardisierte Methoden zur Erstellung von Kernel Threads. Entscheidend im Kontext dieser Arbeit ist es, dass die OpenMP-Threads eigene Ausführungskontexte im Kernel besitzen mit eigener *sched_class*[7], so-

[6] PIDs für Prozesse und TIDs für Threads werden über denselben Mechanismus vergeben [18]. Daher wird teilweise auch allgemein von Task-IDs gesprochen.

[7] Das Struct *sched_class* ist ein Zeiger auf ein Scheduler-Modul pro Thread.

dass sie ganz normal (wie Prozesse) für das Scheduling eingeplant und auch priorisiert werden können.

5.2 OpenMP-Thread-Affinität

Die OpenMP-API bietet bereits eine Möglichkeit, Threads an bestimmte Hardwareressourcen zu binden. Der Konsolenbefehl lscpu gibt einen Überblick über die verfügbaren Sockets, Kerne pro Socket und Threads pro Kern auf einem Linux-System. Ebenfalls können die verfügbaren Nummerierungen ermittelt werden, welche zur Identifikation der Ressource dienen.

Die Anzahl der verfügbaren Hardwareressourcen wird darüber bestimmt, ob der Prozessorkern Simultaneous Multi-Threading (SMT) unterstützt. Wird SMT nicht unterstützt, ist die Anzahl an parallel ausführbaren Hardwarethreads gleich der Anzahl der Prozessorkernen (Nummerierung 0 bis Anzahl Prozessorkerne - 1). Andernfalls können mit SMT auf einem Prozessorkern mehrere Hardwarethreads ausgeführt werden (Nummerierung 0 bis Anzahl Hardwarethreads - 1). [21]

Zum Pinning der OpenMP-Threads an bestimmte Hardwarethreads wird die Funktion *sched_setaffinity()*[8] verwendet, welche die Affinität eines Threads festlegt. Die Affinität ist eine Menge von Hardwarethreads, auf welchen der Thread ausgeführt werden kann. Zur Festlegung dieser wird eine Maske mit der Struktur *cpu_set_t* erstellt, welche die gewünschten Hardwarethreads enthält. [14, 13, 16]

5.2.1 OMP_PLACES

Die Festlegung der Affinität erfolgt über das Setzen der Umgebungsvariable OMP_PLACES, die zum Programmstart eingelesen wird. [21]

Gültige Werte werden durch die Grammatik (siehe Abbildung 11) definiert. Die Grammatik legt dar, dass die Umgebungsvariable entweder durch die Angabe eines abstrakten Namens oder explizit durch die Angabe einer sortierten Liste von kommaseparierten Places erfolgen kann. [21]

Ein Place der Liste ist eine Menge von komma-separierten nicht-negativen Zahlen, umgeben von geschweiften Klammern. Ein Place einer spezifischen Liste wird Place Partition genannt und besitzt ausgehend von der Position der Place Partition in der Liste eine 0-basierte Indexnummer. Zur Angabe einer solchen Liste können auch Intervalle verwendet werden, die nach der Struktur *<lower-bound>:<length>:<stride>* aufgebaut sind. Der Operator *!* negiert die nachfolgenden Angaben und schließt somit Zahlen oder Places aus. [21]

Die Angabe mittels abstrakter Namen erfolgt nach dem Muster *abstract_name(num-places)* mit der optionalen Übergabe von *num-places* zur Angabe der zu erzeugenden Places. Mit dem abstrakten Namen *threads* wird ein Place für jeden Hardwarethread erstellt, was eine feingranulare Aufteilung der Threads auf die verfügbaren Hardwareressourcen ermöglicht. Die Verwendung von *cores* korrespondiert zu einem Place für

[8] Libgomp verwendet die GNU Source Wrapper-Funktionen pthread_setaffinity_np() und pthread_attr_setaffinity_np() zum Setzen der Affinitätsmaske.

einen Prozessorkern, wobei dieser mit SMT mehrere Hardwarethreads zur Verfügung haben kann. Wird der abstrakte Name *ll_caches* verwendet, zeigt jeder Place auf eine Menge von Prozessorkernen, die sich den last level Cache teilen. Eine weitere Möglichkeit gibt die Verwendung von *numa_domains* vor, die Places auf eine Menge von Prozessorkernen mappt, wobei diese sich denselben Speicher mit einer ähnlichen Distanz der verschiedenen Kerne teilen müssen. Non-uniform memory access (NUMA) ist ein Speicherdesign, bei dem auf den eigenen lokalen Speicher eines Prozessorkerns schneller zugegriffen werden kann als auf geteilten Speicher oder lokalen Speicher eines anderen Prozessorkerns. Zuletzt ist die Angabe mit *sockets* möglich, wobei ein Place einem Socket entspricht, welcher mehrere Prozessorkerne enthalten kann. [21, 22]

```
        <list> → <p-list> | <aname>
      <p-list> → <p-interval> | <p-list>,<p-interval>
  <p-interval> → <place>:<len>:<stride> | <place>:<len> |
                 <place> | !<place>
       <place> → {<res-list>} | <res>
    <res-list> → <res-interval> | <res-list>,<res-interval>
<res-interval> → <res>:<num-places>:<stride> |
   <res>:<num-places> | <res> | !<res>
       <aname> → <word>(<num-places>) | <word>
        <word> → sockets | cores | ll_caches | numa_domains |
           threads | <implementation-defined abstract name>
         <res> → non-negative integer
  <num-places> → positive integer
      <stride> → integer
         <len> → positive integer
```

Abbildung 11: Vollständige Grammatik zur Erstellung von gültigen Werten für die OMP_PLACES Umgebungsvariable.

Die folgenden Beispiele erzeugen auf drei verschiedene Arten dieselben Places. In einem System mit vier Hardwarethreads werden die erstellten Threads eines Teams also gleichmäßig auf die Hardwareressourcen verteilt und ein Thread jeweils nur auf einem Place, also einem Hardwarethread ausgeführt.

```
export OMP_PLACES="threads(4)"
export OMP_PLACES="{0},{1},{2},{3}"
export OMP_PLACES="{0}:4:1"
```

Abbildung 12: Verschiedene Beispiele für eine gültige Belegung der OMP_PLACES Umgebungsvariable in einer Bash-Unix-Shell. Alle Beispiele definieren dieselben Places.

5.2.2 OMP_PROC_BIND

Die Umgebungsvariable OMP_PROC_BIND spezifiziert das Pinning-Verhalten innerhalb einer Place-Partition. Wird die Variable auf *false* gesetzt, können OpenMP-Threads

auf andere Places verschoben werden, womit die unter OMP_PLACES definierte Thread-Affinität deaktiviert wird. Ist der Wert *true* wird die Thread Affinität grundsätzlich beachtet und der Initial Thread wird fix an den ersten Place der OMP_PLACES Variable gebunden. [21, 6]

Eine weitere Möglichkeit zum Setzen der Variable besteht darin eine komma-separierte Liste anzugeben, die als gültige Werte *primary*, *close* und *spread* erlaubt. Die Position in der Liste korrespondiert dabei mit der Verschachtelungsebene der Parallel Region. Die Liste sollte also der Verschachtelungstiefe des Programms entsprechen.

Die Primary Regel gibt an, dass alle Threads des Teams auf derselben Place-Partition ausgeführt werden sollen wie der Primary Thread. [21, 6]

Die Close Regel gibt an, dass alle Threads eines Teams nah zu der Place-Partition des Primary Threads platziert werden sollen, wenn es sich um zusammenhängende Place-Partitionen handelt.

Die Spread Regel gibt an, dass die Threads eines Teams über alle Place-Partitionen gleichmäßig verteilt werden. [21, 6]

5.2.3 Pinning mit Umgebungsvariablen

Mittels den bereits durch die OpenMP-API vorgegebenen Spezifikationen ist es möglich ein implizites Pinning vorzunehmen. Dies ist jedoch lediglich für das For-Workshare-Konstrukt möglich. Dazu muss das Scheduling statisch (nach dem Round-Robin Prinzip) erfolgen und die Anzahl an Threads auch konkret (entweder über die Umgebungsvariable OMP_NUM_THREADS oder den Code[9]) angegeben werden. Wenn die Implementationsspezifika betrachtet werden, dass die Places chronologisch den Threads zugeordnet werden[10] und dem Primary Thread stets Id 0 zugeordnet wird, dann kann hierrüber ein Pinning erfolgen. Da das Round Robin Scheduling ebenfalls die Threadreihenfolge beachtet, kann über die Chunkgröße entschieden werden, welcher Thread (mit entsprechendem Place) welchen Indexteil der For-Schleife ausführt. Wird die Chunkgröße zum Beispiel auf 1 fixiert und eine For-Schleife mit 4 Schleifenzyklen auf 4 Threads statisch ausgeführt, bearbeitet der erste Primary Thread den ersten Chunk und die weiteren Worker Threads chronologisch die übrigen 3 Chunks (siehe Abbildung 13). Das Pinning erfolgt also implizit durch ein Mapping der Reihenfolge der Threads auf die Reihenfolge der Schleife. [21, 22]

Mit der OpenMP-Version 5.1 (ab gcc 12) kann dieses reproduzierbare Verhalten auch durch Angabe einer *order*-Klausel erfolgen. Wird das For-Pragma um die *order*-Klausel mit der Angabe von *reproducible* erweitert, so entspricht das Verhalten dem einer statischen For-Schleife (siehe Abbildung 13). [21, 22]

Aus dieser Konstellation ergeben sich aber einige Probleme für den Einsatz im planungsbasierten Scheduler. Da das implizite Pinning nur für ein Workshare-Konstrukt

[9] Die Angabe von num_threads(n) ist nur für parallel gültig und muss auf die umgebende Parallel Region gesetzt werden.

[10] Dies gilt explizit nur für libgomp, da die API nicht spezifiziert auf welchem Place der Primary Thread initial startet. Libgomp wiederum fixiert diesen stets an den ersten Place.

möglich ist, wäre der Einsatz der OpenMP-Direktiven massivst eingeschränkt. Parallel-Konstrukte und Sections können nicht implizit gepinnt werden. Weiterhin kann zwar von außen die Placelist bearbeitet werden, um ein geändertes Pinning zu ermöglichen, jedoch stößt dieser Mechanismus schnell an seine Grenzen. Werden Places mehrfach angegeben, um die Reihenfolge des impliziten Pinnings zu erweitern, wird für jeden Place auch ein Thread erstellt (insofern die Threadanzahl nicht schon im Code fix angegeben wurde). Dadurch entsteht ein nicht gewollter Overhead für den Pinningmechnismus. Die einzige Möglichkeit das Pinningverhalten nach der Kompilierung zuverlässig zu ändern besteht in der Anpassung des Programmcodes und einer erneuten Kompilierung, was für den Einsatz mit dem planungsbasierten Scheduler nicht optimal ist. Der planungsbasierte Scheduler sollte außerdem mit dem Programm kommunizieren können, was das implizite Pinning nicht ermöglicht.

```
#pragma omp parallel
{
  #pragma omp for schedule(static, 1)
  for (int i = 0; i < 4; i++)
  {
    // code scope of parallel executed implicit tasks
  }
}

#pragma omp parallel
{
  #pragma omp for order(reproducible: concurrent)
  for (int i = 0; i < 4; i++)
  {
    // code scope of parallel executed implicit tasks
  }
}
```

Abbildung 13: Beispiele für implizites Pinning in reproduzierbaren For-Schleifen.

5.3 Task-Affinität

Im Kontext von expliziten Tasks bietet OpenMP die Möglichkeit an, Tasks mit der optionalen Klausel *affinity()* zu versehen. In die Klammer wird dann eine Variable übergeben, um anzuzeigen, dass die Ausführung des Tasks „nah“ zu dem physischen Speicher der Variable erfolgen soll. Zum Beispiel kann bei einer 2 Socket Plattform mit lokalen Speicherkomponenten (NUMA) unterschieden werden auf welchem Speicher die Variablen gespeichert sind und dann den mit *affinity()* spezifizierten Task auf einem entsprechenden Thread des Sockets ausführen. [21]

Die Funktionalität dieser Funktion ist also abhängig von der Architektur des Systems und bietet sich daher eher weniger für ein zuverlässiges Pinning an. Zudem ist der Programmfluss bei Verwendung von Tasks weniger planbar, da die Tasks zu unabhängigen Zeitpunkten ausgeführt werden können, was das planungsbasierte Scheduling schwieriger macht.

6 Prototypische Entwicklung eines Pinning-Mechanismus

Wie aus dem vorherigen Kapitel zu entnehmen ist, bietet die OpenMP-API bereits über die OpenMP-Thread Affinität feingranulare Möglichkeiten die Threads auf die verfügbaren Hardwareressourcen zu verteilen, jedoch kann nur bedingt gesteuert werden, welcher Place-Thread einen bestimmten Task ausführt. Die folgenden Kapitel betrachten daher die Implementationsspezifika genauer und erörtern die Idee eines weitergehenden Pinning-Mechanismus für Threads an bestimmte Places. Die Idee der Places soll hierbei beibehalten werden, sodass eine feine Unterteilung der Hardwareressourcen weiterhin möglich ist.

Der Prototyp fokussiert sich auf das Pinning von spezifischen Section-Threads zu Places, da hier die Last potenziell am unausgeglichensten ist und dies das Pinning auf bestimmte Hardwarethreads interessanter macht.

6.1 Compiler Übersetzung

Die OpenMP-Implementierung übersetzt die Workshare-Direktiven (for und section) in Schleifen, welche nach jeder Iteration einen neuen Workshareteil zugewiesen bekommen. Die in Abbildung 14 gezeigte Compiler Übersetzung einer Section-Direktive zeigt die For-Schleife mit dem Startindex der Rückgabe der Funktion *GOMP_sections_start()* und fortlaufenden Indizes der Rückgabe der Funktion *GOMP_sections_next()*. Die Abbruchbedingung der Schleife tritt ein, wenn die Funktion *GOMP_sections_next()* 0 zurückgibt und damit die Arbeit des Threads beendet ist. Programmintern rufen beide Funktionen *gomp_iter_dynamic_next()* (aus iter.c) auf, welche die zu bearbeitende Section als Rückgabewert zurückgibt oder 0 wenn alle Workchunks bearbeitet wurden. Die Position der ursprünglichen Section im Programmcode korreliert direkt zur Position der Switch Anweisung, wobei der Index 1-basiert ist. [6]

6.2 direktes Pinning

Die naheliegendste Möglichkeit, ein explizites Pinning direkt mit der Erstellung eines Threads vorzunehmen, ist nicht möglich. Dabei würde es sich grundsätzlich anbieten, in den *pthread_create()* Syscall (siehe Abbildung 15) direkt die passende Funktion für den Thread mit dem passenden Place zu übergeben.

Das ist aber nicht möglich, da auch das *#pragma omp sections* in eine Schleife übersetzt wird und sich somit niemals die Funktion der ausführenden Threads unterscheidet, obwohl pro Section potenziell komplett unterschiedlicher Code ausgeführt wird.

Zudem verwenden OpenMP-Konstrukte wie *#pragma omp parallel* und *#pragma omp for* per Definition denselben Programmcode und unterscheiden sich daher lediglich darin, wie oft eine Funktion parallel ausgeführt wird bzw. welchen Bereich der Schleife sie ausführen sollen, aber nicht in der tatsächlich ausgeführten Funktion. Hier bietet es sich potenziell auch an, die in Kapitel 5.2.1 und 5.2.2 diskutierten Umgebungsvariablen zu nutzen, da der Algorithmus hierbei in der Regel in gleich große Teile unterteilt wird und somit uninteressanter für ein weiteres Pinning macht.

```
// Das Section-Pragma wird ...
#pragma omp sections
{
  #pragma omp section
  stmt1;
  #pragma omp section
  stmt2;
  #pragma omp section
  stmt3;
}

// ... in eine for Schleife mit switch Statement uebersetzt
for (i = GOMP_sections_start (3); i != 0; i =
    GOMP_sections_next ())
{
  switch (i)
  {
    case 1:
    stmt1;
    break;
    case 2:
    stmt2;
    break;
    case 3:
    stmt3;
    break;
  }
}

GOMP_barrier ();
```

Abbildung 14: Gcc Compiler-Übersetzung einer Section in eine For-Schleife mit Switch-Anweisung. [6]

6.3 indirektes Pinning

Der gewählte Ansatz des Pinnings ergibt sich aus dem Laufzeitverhalten der OpenMP-Umgebung. Da die Sections nach der Kompilation ebenfalls mit einer Schleife ausgeführt werden, kann zur Laufzeit entschieden werden, welcher Chunk des Workshare-Konstrukts als nächstes ausgeführt werden soll. Es besteht also die Möglichkeit, an dieser Stelle den aktuellen Place des Threads zu überprüfen und anhand dessen zu entscheiden, welcher Chunk als nächstes bearbeitet werden soll. Der Eingriff in die Vergabe des nächsten Chunks erfolgt hierbei in der *gomp_iter_dynamic_next()* Funktion.

Eine entscheidende Voraussetzung für diese Art des Pinnings ist, dass alle definierten Places auch tatsächlich von Threads des Teams ausgeführt werden. Diese Garantie wird durch die OpenMP-Umgebung bei Erstellung eines Teams gegeben[11], da diesel-

[11] Wenn die Anzahl an Threads größer gleich die Anzahl an Places ist.

```
// gcc/libgomp/team.c: Zeile 858
err = pthread_create(&start_data->handle, attr, gomp_thread_start,
    start_data);
```

Abbildung 15: Aufruf der POSIX Funktion p_thread_create zur Erstellung neuer Threads eines Teams.

be Anforderung bereits für die Threadverteilung auf die Places über die Umgebungsvariable *OMP_PLACES* besteht. Sind die gewünschten Places also in der Variable angegeben, werden sie auch für neue Teams zur Laufzeit beachtet.

6.3.1 Textdatei

Das Pinning wird über eine Konfigurationsdatei ermöglicht, welche im Arbeitsverzeichnis unter dem Namen *thread_pinning.txt* abgelegt wird. Die Datei ist nach dem Schema Place und Section Places aufgebaut, d.h. der entsprechende Thread mit dem angegebenen Place soll die darauffolgend aufgelisteten Sections ausführen.

Gibt es Places mehrmals, ist damit implizit die nächste Parallel Region gemeint. Die Reihenfolge der Datei muss daher auch dem Programmfluss entsprechen. Die Beispieldatei aus Abbildung 16 gibt also das Pinning für ein Programm mit zwei Parallel Sections mit jeweils 8 Section Elementen vor. Bei einer Ausführung mit 4 Threads werden die Sections also gleichmäßig auf die Threads verteilt. Zu beachten ist, dass weiterhin die Umgebungsvariable OMP_PLACES gesetzt werden muss, um die benutzbaren Places anzugeben. Der Place aus der Konfigurationsdatei ist also nur ein Verweis auf den ersten Eintrag (1-basiert) aus der OMP_PLACES Variable.

```
# [Place] [Section Places]
0 7,6
1 5,4
2 3,2
3 1,0
#
0 0,1
1 2,3
2 4,5
3 6,7
```

Abbildung 16: Konfigurationstextdatei zum Pinnen von beliebig vielen Section-Places auf einen Thread-Place. Das Beispiel zeigt ein Pinning mit zwei aufeinanderfolgenden Sections und vier ausführenden Threads.

6.3.2 Implementierung

Zunächst wird die Konfigurationsdatei bei Programmstart eingelesen, nachdem die Umgebungsvariable *OMP_PLACES* geparsed wurde (parse_places_var() in env.c). So kann sichergestellt werden, dass die Datei nur dann eingelesen wird, wenn auch die

notwendigen Places definiert wurden. Die definierten Pinnings werden für den weiteren Ablauf mittels der *structs place_pinning* und *chunk* gespeichert (siehe Abbildung 17). Dabei enthält ein *place_pinning* jeweils einen Place, ein Flag ob das Pinning bereits verwendet wurde, einen Zeiger auf die zugewiesenen Chunks und einen Zeiger auf den nächsten Place in derselben Reihenfolge wie die Konfigurationsdatei. Das *chunk struct* spiegelt die 1:N Beziehung zwischen Place und Section Places wider, da pro Place eine Liste von Section Places angehängt werden kann. Zudem existiert ein Zeiger auf das Vater *place_pinning struct* und ein *next_alloc* Zeiger verweist auf das nächste Section Place Element in der Reihenfolge der Datei.

```
struct place_pinning
{
  int place;
  bool done;
  struct chunk *chunks;
  struct place_pinning *next_alloc;
};

struct chunk
{
  struct place_pinning *current_place;
  int chunk_num;
  struct chunk *next_alloc;
};
```

Abbildung 17: Structs aus gcc/libgomp/libgomp.h zur Speicherung der Textdateiinhalte.

Die bereits erwähnte Verteilung in *gomp_iter_dynamic_next()* (siehe Abbildung 22) sucht zunächst den ersten passenden Place Eintrag, welcher noch nicht als abgearbeitet markiert wurde. Um die Laufzeit von O(n) für weitere Iterationen zu optimieren, wird ein Zeiger auf den aktuellen Section Place zu dem Thread gespeichert. Nach der ersten Iteration im Thread kann die nächste Section in konstanter Zeit ermittelt werden. Sind alle Section Places von einem Thread bearbeitet, wird der Eintrag als bearbeitet (*done-Flag*) markiert, sodass spätere Teams aus neuen Parallel Regions diesen ignorieren.

7 Ergebnisse

Der Unterschied zwischen dem Section Placing über Umgebungsvariablen und dem Pinning mit der prototypischen Entwicklung wird anhand der Betrachtung eines Beispiels deutlich (siehe Abbildung 23). Dabei wurden die folgenden Umgebungsvariablen für das Placing verwendet.

```
export OMP_PLACES="{0},{1},{2},{3}"
export OMP_PROC_BIND=true
export OMP_NUM_THREADS=4
```

Abbildung 18: Konfiguration der Umgebungsvariablen des Testprogramms.

Zusätzlich liegt bezüglich der prototypischen Ausführung eine Textdatei vor, die das folgende Ausführungsmodell widerspiegelt.

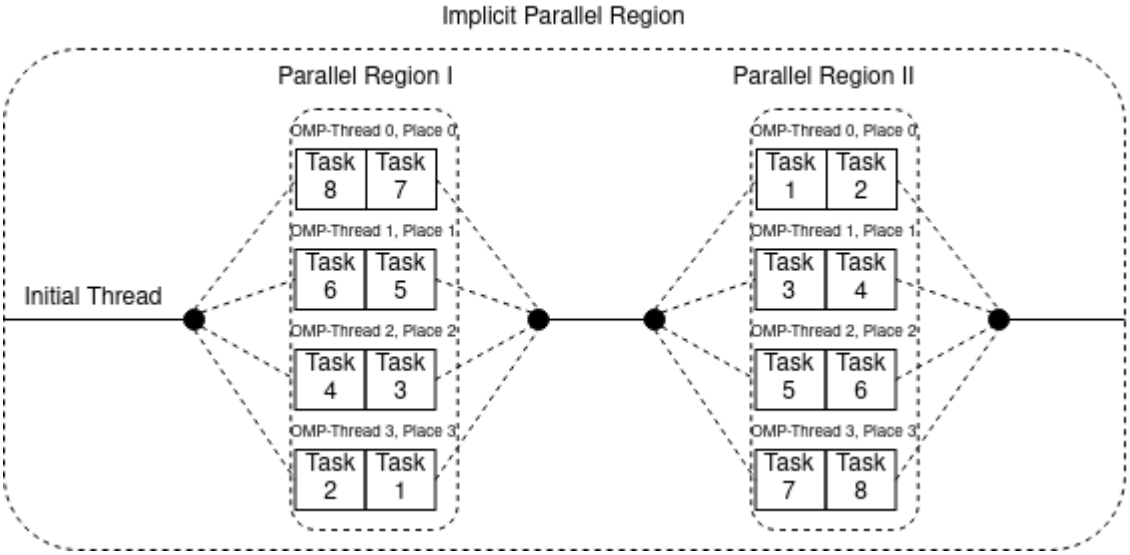

Abbildung 19: Fork-Join-Modell zur Laufzeit des Beispielprogramms.

Während die Original-Implementierung eine möglichst passende Aufteilung unter den Places anstrebt, folgt die prototypische Entwicklung strikt den gewünschten Angaben aus der Konfigurationsdatei. Zu erkennen ist, dass die Original-Implementierung eine möglichst gleichmäßige Aufteilung unter den Threads erzielt (siehe Abbildung 24). Es ist zu erkennen, dass ein Thread nicht genau 2 Sections ausführt, jedoch kann dies auch an der Testumgebung liegen, welche nicht exklusiv auf dem Testgerät ausgeführt wird. Weiterhin ist sichtbar, dass keinerlei Konsistenz bezüglich eines Pinnings zu Threads besteht, da keine bestimmte Reihenfolge des Thread-Scheduling in Bezug auf die Section Places erfolgt.

Ein konsequentes Pinning wird mit der prototypischen Entwicklung ermöglicht (siehe Abbildung 25), welche exakt die Vorgaben der Textdatei aus Kapitel 6.3.1 umsetzt. Die Reihenfolge der Threadausführung ist hierbei limitiert auf die Reihenfolge der Sections innerhalb eines Threads.

Abseits des impliziten Pinnings für For-Schleifen kann das explizite indirekte Pinning analog zu der Implementierung für die Sections für For-Schleifen übernommen werden. Hierbei sind die Funktionen GOMP_loop_runtime_start() und GOMP_loop_runtime_next() für ein dynamisches Scheduling relevant und GOMP_loop_static_next() für ein statisches Scheduling. Das Pinning für Sections kann jedoch nicht 1:1 für Parallel Regions übernommen werden, da diese lediglich einmal vor der Ausführung gescheduled werden. In der statischen Startfunktion, die alle Threads zunächst ausführen, könnte hier ein Pinning erfolgen.

8 Einschränkungen

Der Prototyp bezieht sich bis jetzt auf einen Programmfluss, dessen Parallel Regions nicht geschachtelt sein dürfen. Mögliche Erweiterungen für die Lösung dieses Problems wären, entweder die Dateistruktur der Konfigurationsdatei so anzupassen, dass sie eine Baumstruktur verwendet oder eine globale eindeutige Thread-Id für die OpenMP-Implementierung einzuführen. Die globale Id könnte anhand einer statisch inkrementierten Variable vergeben werden, wobei die Eindeutigkeit durch die zugesicherte Reihenfolge der Threaderstellung einer Parallel Region gegeben ist.

Je nach Hardwareunterstützung kann die OpenMP-Umgebung auf atomare Befehle zurückgreifen und somit nur kleinere Abschnitte durch Locking-Mechanismen blockieren (siehe Abbildung 20). Da diese Unterstützung auf der Testumgebung gegeben ist, wurde der Prototyp aufbauend auf diesen atomaren Operationen aufgebaut. Die Implementierung ohne Hardware unterstützte Synchronisierungsmechanismen ist daher noch ausstehend und wird von dem Prototyp nicht unterstützt. [17]

```
#ifdef HAVE_SYNC_BUILTINS

if (gomp_iter_dynamic_next(&s, &e))
  ret = s;
else
  ret = 0;

#else
struct gomp_thread *thr = gomp_thread();

gomp_mutex_lock(&thr->ts.work_share->lock);
if (gomp_iter_dynamic_next_locked(&s, &e))
  ret = s;
else
  ret = 0;
gomp_mutex_unlock(&thr->ts.work_share->lock);

#endif
```

Abbildung 20: Programmausschnitt mit *#ifdef HAVE_SYNC_BUILTINS*, wobei der Programmfluss je nach Wert auf einen unterschiedlichen Zuweisungsmechanismus zurückgreift.

9 Fazit

Grundsätzliche eignet sich OpenMP sehr gut für den Einsatz in Grid-Systemen mit planungsbasierten Scheduler, da die OpenMP-Implementierung ein 1:1 Threading-Modell umsetzt und somit jedem OpenMP-Thread ein Kernelthread zugeordnet wird. Damit kann der planungsbasierte Scheduler des Computing Nodes auch exakt die Zeiträume planen, in denen die OpenMP-Threads ausgeführt werden.

Die bestehenden Möglichkeiten des Pinnings und der Verwaltung aller OpenMP-Threads sind vor allem durch die Konfiguration der Places gegeben. Hierbei bietet aber nur die Angabe der Places in Kombination mit der For-Direktive die Möglichkeit eines impliziten Pinnings. Zudem kann das Pinning von außen nur bedingt angepasst werden, sodass potenziell eine Änderung des Programmcodes nötig wird, wenn das Pinning angepasst werden soll.

Die Lösung wird hierfür exemplarisch anhand des Prototypen für Section-Direktiven skizziert und ist auch auf weitere Direktiven übertragbar. So ist es möglich, dass nach einer eingehenden Programmanalyse die Konfigurationsdatei geschrieben wird und auch auf geänderte Problemgrößen schnell angepasst werden kann. Eine automatisierte Erstellung der Datei ist hier denkbar. Der Austausch mit der Grid Middleware erfolgt hierbei über die Datei, da durch das Ablegen der Datei im Arbeitsverzeichnis direkt das Pinning erfolgt. Zu beachten ist hier auch, dass die Umgebungsvariablen vor Programmstart gesetzt werden müssen, damit das gewünschte Pinning erfolgt.

10 Ausblick

Die weiteren Arbeitsschwerpunkte sollten zunächst die Integration des Prototyps in den Kontext eines planungsbasierten HPC-Systems bringen, sodass die Kommunikation zwischen Middleware und HPC-System getestet werden kann.

Zudem muss eine vollständige Implementierung des Pinning Mechanismus auf alle OpenMP-Direktiven übertragen werden und somit eine vollständige Funktionalität der libgomp Implementierung in gcc angestrebt werden. Noch ausstehend ist zudem, wie sich die Verwendung des Threadpools auf das Monitoring der Grid Middleware auswirkt. Hierbei muss noch untersucht werden, inwiefern die Idle-Threads möglicherweise die Analyse verfälschen. Dazu sollte das Monitoring in der Lage sein, die Idle-Threads rauszufiltern.

Literatur

[1] Gene M. Amdahl. “Validity of the single pro cessor approach to achieving large scalecomputing capabilities”. In: *AFIPS spring joint computer conference* (1967). URL: https://www-inst.eecs.berkeley.edu//~n252/paper/Amdahl.pdf.

[2] Raghu Bharadwaj. *Mastering Linux Kernel development : a kernel developer's reference manual / Raghu Bharadwaj*. eng. 1st edition. Birmingham, England ; Packt Publishing, 2017. ISBN: 1-78588-613-4.

[3] Ulrich Drepper und Ingo Molnar. “The Native POSIX Thread Library for Linux”. In: (2003). URL: https://static.redhat.com/legacy/whitepapers/developer/POSIX_Linux_Threading.pdf.

[4] I. Foster und C. Kesselman. *The grid: Blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558-604-758.

[5] Kelvin Glaß. “Plan Based Thread Scheduling on HPC Nodes”. Magisterarb. Freie Universität Berlin, Institut für Informatik, 2018.

[6] *GNU Offloading and Multi Processing Runtime Library*. 51 Franklin Street, Fifth Floor Boston, MA 02110-1301, USA: Free Software Foundation, Inc, 2023. URL: https://gcc.gnu.org/onlinedocs/gcc-13.1.0/libgomp.pdf.

[7] Ananth Grama und Ahmed H. Sameh, Hrsg. *Parallel Algorithms in Computational Science and Engineering*. eng. Cham, 2020.

[8] Matthias Hovestadt u. a. “Scheduling in HPC Resource Management Systems: Queuing vs. Planning”. In: *Job Scheduling Strategies for Parallel Processing*. Hrsg. von Dror Feitelson, Larry Rudolph und Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, S. 1–20. ISBN: 978-3-540-39727-4.

[9] Haoqiang Jin u. a. “High performance computing using MPI and OpenMP on multi-core parallel systems”. In: *Parallel Computing* 37.9 (2011). Emerging Programming Paradigms for Large-Scale Scientific Computing, S. 562–575. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0167819111000159.

[10] Michael Kerrisk, Hrsg. *clone(2) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man2/clone.2.html.

[11] Michael Kerrisk, Hrsg. *futex(7) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/futex.7.html.

[12] Michael Kerrisk, Hrsg. *nptl(7) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/nptl.7.html.

[13] Michael Kerrisk, Hrsg. *pthread_attr_setaffinity_np(3) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man3/pthread_attr_setaffinity_np.3.html.

[14] Michael Kerrisk, Hrsg. *pthread_setaffinity_np(3) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man3/pthread_getaffinity_np.3.html.

[15] Michael Kerrisk, Hrsg. *pthreads(7) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html.

[16] Michael Kerrisk, Hrsg. *sched_setaffinity(2) - Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

[17] *Legacy __sync Built-in Functions for Atomic Memory Access*. Free Software Foundation, Inc, 2023. URL: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html.

[18] Sandra Loosemore. *The GNU C Library Reference Manual*. 2023. URL: https://www.gnu.org/software/libc/manual/2.37/pdf/libc.pdf (besucht am 23. 07. 2023).

[19] Rachel McDowell. *Science at Exascale: The Future of Fusion Modeling*. URL: https://www.olcf.ornl.gov/2019/05/07/science-at-exascale-the-future-of-fusion-modeling/ (besucht am 10. 08. 2023).

[20] John Michalakes. "HPC for Weather Forecasting". In: *Parallel Algorithms in Computational Science and Engineering*. Hrsg. von Ananth Grama und Ahmed H. Sameh. Cham: Springer International Publishing, 2020, S. 297–323. ISBN: 978-3-030-43736-7. DOI: 10.1007/978-3-030-43736-7_10. URL: https://doi.org/10.1007/978-3-030-43736-7_10.

[21] *OpenMP Application Programming Interface*. OpenMP Architecture Review Board, Nov. 2021. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[22] *OpenMP Application Programming Interface Examples*. OpenMP Architecture Review Board, 2022. URL: https://www.openmp.org/wp-content/uploads/openmp-examples-5.2.1.pdf.

[23] *OpenMP Compilers & Tools*. OpenMP Architecture Review Board. 2023. URL: https://www.openmp.org/resources/openmp-compilers-tools/#compilers (besucht am 10. 08. 2023).

[24] E. Strohmaier, J. Dongarra und H. Simon. *Top 500 list of the 500 fastest supercomputers*, 2023. URL: https://www.top500.org/lists/top500/2023/06/ (besucht am 10. 08. 2023).

[25] Andrew Tanenbaum und Herbert Bos. *Modern Operating Systems, Global Edition*. eng. Harlow, United Kingdom: Pearson Higher Education & Professional Group, 2014. ISBN: 9781292061429.

[26] *What is ITER?* ITER International Fusion Energy Organization. URL: https://www.iter.org/proj/inafewlines (besucht am 10. 08. 2023).

Abbildungsverzeichnis

A Anhang

```
// Backtrace eines Beispiel OpenMP Programms mit gdb glibc 2.37
    und gcc 13:
#0  clone3 () at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:42
#1  0x0000000000049ef81 in __clone_internal (cl_args=
    cl_args@entry=0x7fffffffd330, func=func@entry=0x496780 <
    start_thread>, arg=arg@entry=0x7ffff7ff86c0)
at ../sysdeps/unix/sysv/linux/clone-internal.c:54
#2  0x0000000000496695 in create_thread (pd=pd@entry=0
    x7ffff7ff86c0, attr=attr@entry=0x4edbe0 <gomp_thread_attr>,
    stopped_start=stopped_start@entry=0x7fffffffd416,
stackaddr=stackaddr@entry=0x7ffff77f8000, stacksize=<optimized
    out>, thread_ran=thread_ran@entry=0x7fffffffd417) at
    pthread_create.c:297
#3  0x00000000004970b4 in __pthread_create_2_1 (newthread=
    newthread@entry=0x7fffffffd578, attr=attr@entry=0x4edbe0 <
    gomp_thread_attr>,
start_routine=start_routine@entry=0x475390 <gomp_thread_start>,
    arg=arg@entry=0x7fffffffd500) at pthread_create.c:833
#4  0x0000000000475b81 in gomp_team_start (fn=fn@entry=0x46b643
    <main._omp_fn.0>, data=data@entry=0x0, nthreads=
    nthreads@entry=4, flags=flags@entry=0, team=0x4f0d40,
    taskgroup=taskgroup@entry=0x0)
at ../../../../gcc/libgomp/team.c:858
#5  0x000000000046c57e in GOMP_parallel_sections (fn=0x46b643 <
    main._omp_fn.0>, data=0x0, num_threads=4, count=3, flags=0)
    at ../../../../gcc/libgomp/sections.c:237
#6  0x000000000046b63c in main () at example_program.c:34
```

Abbildung 21: Vollständiger Backtrace der Threaderstellung eines OpenMP-Programms.

```
// only use place pinning when places have been defined
if (__builtin_expect(gomp_places_list != NULL, 0))
{
  // when chunk pointer not found search it initially
  if (thr->current_chunk == NULL)
  {
    explicit_place_pinning p;
    p = head_explicit_place_pinning;

    // iterate until matching place
    while (p != NULL)
    {
      // extern placeindex is 0-based; intern index 1-based
      if (p->place + 1 == thr->place && p->done == false)
      {
        thr->current_chunk = p->chunks;
        break;
      }
      p = p->next_alloc;
    }
  }
  else if (thr->current_chunk->current_place->done == true)
  {
    // all chunks have been executed; now finally break out
       scheduling
    thr->current_chunk = NULL;
    return false;
  }

  // add 1 because of different offset
  tmp = thr->current_chunk->chunk_num + 1;

  if (thr->current_chunk->next_alloc == NULL)
  {
    // work is done; but gets scheduled another time
    thr->current_chunk->current_place->done = true;
  }
  else
  {
    // allocate next chunks for next scheduling round
    thr->current_chunk = thr->current_chunk->next_alloc;
  }
}

// safety check
if (tmp >= end)
  return false;

nend = tmp + chunk;
if (nend > end)
  nend = end;
*pstart = tmp;
*pend = nend;
return true;
```


Abbildung 22: Codeausschnitt aus iter.c zum Pinnen von Section-Threads an Places. (Link zum Sourcecoderepository mit vollständigem Code: `https://git.imp.fu-berlin.de/erdweh97/libgomp-pinning`)

```c
#include <stdio.h>
#include <omp.h>

void work(int id)
{
  // execute fibonacci calculation for workload
  long t1 = 0, t2 = 1, nextTerm = 1;

  for (int i = 0; i <= 1000000; i++)
  {
    t1 = t2;
    t2 = nextTerm;
    nextTerm = t1 + t2;
  }

  printf("OMP-Thread %d at place %d executed section %d \n",
     omp_get_thread_num(), omp_get_place_num(), id);
}

int main()
{
  for (int i = 0; i < 2; i++)
  {
    printf("-----------------------------------------------\n");
    printf("Start of Parallel Section %d\n", i + 1);
    printf("-----------------------------------------------\n");

    #pragma omp parallel sections
    {
      #pragma omp section
      work(0);
      #pragma omp section
      work(1);
      #pragma omp section
      work(2);
      #pragma omp section
      work(3);
      #pragma omp section
      work(4);
      #pragma omp section
      work(5);
      #pragma omp section
      work(6);
      #pragma omp section
      work(7);
    }

    printf("-----------------------------------------------\n");
    printf("End of Parallel Section %d\n", i + 1);
    printf("-----------------------------------------------\n");
    printf("\n");
  }

  return 0;
}
```


Abbildung 23: Testprogramm zum Testen des Pinningverhaltens. (Link zum Sourcecoderepository mit vollständigem Code: `https://git.imp.fu-berlin.de/erdweh97/libgomp-pinning`)

```
-------------------------------------------
Start of Parallel Section 1
-------------------------------------------
OMP-Thread 3 at place 3 executed section 3
OMP-Thread 0 at place 0 executed section 2
OMP-Thread 3 at place 3 executed section 4
OMP-Thread 1 at place 1 executed section 0
OMP-Thread 0 at place 0 executed section 5
OMP-Thread 3 at place 3 executed section 6
OMP-Thread 2 at place 2 executed section 1
OMP-Thread 1 at place 1 executed section 7
-------------------------------------------
End of Parallel Section 1
-------------------------------------------

-------------------------------------------
Start of Parallel Section 2
-------------------------------------------
OMP-Thread 0 at place 0 executed section 3
OMP-Thread 3 at place 3 executed section 2
OMP-Thread 0 at place 0 executed section 4
OMP-Thread 3 at place 3 executed section 5
OMP-Thread 2 at place 2 executed section 1
OMP-Thread 0 at place 0 executed section 6
OMP-Thread 1 at place 1 executed section 0
OMP-Thread 3 at place 3 executed section 7
-------------------------------------------
End of Parallel Section 2
-------------------------------------------
```

Abbildung 24: Ausgaben des Testprogramms mit der originalen gcc libgomp Implementierung.

```
-------------------------------------------
Start of Parallel Section 1
-------------------------------------------
OMP-Thread 2 at place 2 executed section 3
OMP-Thread 0 at place 0 executed section 7
OMP-Thread 3 at place 3 executed section 1
OMP-Thread 3 at place 3 executed section 0
OMP-Thread 2 at place 2 executed section 2
OMP-Thread 1 at place 1 executed section 5
OMP-Thread 0 at place 0 executed section 6
OMP-Thread 1 at place 1 executed section 4
-------------------------------------------
End of Parallel Section 1
-------------------------------------------

-------------------------------------------
Start of Parallel Section 2
-------------------------------------------
OMP-Thread 2 at place 2 executed section 4
OMP-Thread 0 at place 0 executed section 0
OMP-Thread 3 at place 3 executed section 6
OMP-Thread 2 at place 2 executed section 5
OMP-Thread 3 at place 3 executed section 7
OMP-Thread 1 at place 1 executed section 2
OMP-Thread 0 at place 0 executed section 1
OMP-Thread 1 at place 1 executed section 3
-------------------------------------------
End of Parallel Section 2
-------------------------------------------
```

Abbildung 25: Ausgaben des Testprogramms mit der prototypischen gcc libgomp Implementierung.