Abstraktionen zur performanten Programmierung von Multi-Core Architekturen mit hierarchischem Speicher[1]

Christian Terboven[2]

Abstract: Parallelprogrammierung für Computer mit gemeinsamem Speicher (Shared Memory) erscheint oftmals einfach. Allerdings hängt die für Anwendungen erreichbare Leistung von bestimmten Eigenschaften der Systemarchitektur ab. Diese Arbeit präsentiert Lösungen um Shared Memory-parallele Anwendungen mittels eines methodischen Ansatzes für aktuelle und zukünftige Architekturen auszulegen. Geeignete Abstraktionen für Parallelität und Lokalität müssen, um erfolgreich zu sein, gleichzeitig leistungsfähig und einfach in der Verwendung in bestehendem Programmcode sein. Dabei kommen dem Begriff „Abstraktion" in dieser Arbeit zwei Bedeutungen zu. Zum einen ist damit die methodische Auswahl von berücksichtigungswürdigen Architektureigenschaften gemeint. Zum anderen wird hierunter der Entwurf von Konzepten und Mustern für die Parallelprogrammierung zur Unterstützung der Softwareentwicklung verstanden.

1 Einleitung

Bis vor Kurzem hat sich die Ausführungsgeschwindigkeit von seriellen Programmen auf einem jeweils aktuellen Prozessor etwa alle 18 Monate verdoppelt. Wie Asanovic et al. beobachtet haben, haben Softwareentwickler die kontinuierliche Erhöhung der Leistungsaufnahme der Prozessoren akzeptiert, solange diese mit der Erhöhung der Ausführungsgeschwindigkeit serieller Programme einherging [As09]. Diese Entwicklung endete in 2009. Seitdem erhöht sich zwar die Integrationsdichte und die absolute Leistung von Prozessoren weiterhin, aber diese Steigerung resultiert vor allem aus der Erhöhung der Anzahl der Rechenkerne (Cores), die zu Multi-Core Architekturen geführt hat. Allerdings kann die Leistungssteigerung jetzt nur noch durch aktive Ausnutzung der Parallelität abgerufen werden.

Der natürliche Ansatz zur Ausnutzung von Multi-Core Parallelität besteht im Einsatz von Threads. Dabei hat sich OpenMP zum de-facto Standard der Shared-Memory Parallelprogrammierung entwickelt und findet vor allem im Hochleistungsrechnen Verwendung. OpenMP besteht aus einem Satz von Direktiven (Pragmas) zum Ausdruck von Parallelität im Programmcode, sowie Laufzeitfunktionen und Umgebungsvariablen zur Beeinflussung der Ausführung eines OpenMP Programms. OpenMP unterstützt sowohl das klassische Worksharing, z.B. die Aufteilung von Iterationen einer `for`-Schleife auf mehrere Threads, als auch das Tasking, den Ausdruck von feingranularen nebenläufigen Ausführungselementen.

[1] Englischer Titel der Dissertation: "Abstractions for Performance Programming on Multi-Core Architectures with Hierarchical Memory"

[2] RWTH Aachen University, terboven@itc.rwth-aachen.de

Zur Erhöhung der Speicherbandbreite ist in modernen Prozessoren der Speichercontroller integriert und damit bei Systemen mit zwei oder mehr Prozessoren der vorhandene physikalische Speicher partitioniert: Zugriffe eines Rechenkerns auf direkt angebundenen Speicher erfolgen mit geringerer Latenz und höherer Bandbreite als auf entfernten Speicher. Dieses Verhalten bezeichnet man als NUMA (non-uniform memory access). Parallele Programme zeigen folglich auf NUMA-Architekturen schlechte Leistung, wenn auf entfernt liegende Daten zugegriffen wird, und Task-parallele Programme bedürfen Methoden zur Zuordnung von Tasks zu Daten oder Threads. Dabei vertieft sich die Speicherhierarchie durch mehrere Cache-Ebenen, mehrere Speichercontroller pro Prozessor, sowie das Aufkommen neuer Technologien, wie z.B. Speicher mit sehr hoher Bandbreite oder nichtflüchtigem Speicher. Somit wird für hohe Energieeffizienz und Ausführungsleistung die Optimierung des Datenzugriffs notwendig. Dazu bedarf es geeigneter Abstraktionen um die Programmierbarkeit und Wartbarkeit der Codes zu erhalten. Als wissenschaftliche Beiträge liefert diese Arbeit [Te16]:

- Die Definition und Diskussion von abstrakten Ausdrucksweisen zur Speicherverwaltung auf NUMA-Architekturen.
- Die Entwicklung des flexiblen und leistungsfähigen *Affinity Modell*, das in den Standard OpenMP 4.0 eingeflossen ist.
- Die Weiterentwicklung und Bewertung von Methoden zur Parallelisierung Objektorientierter C++-Anwendungen sowie die Unterstützung des Affinity Modells durch den Einsatz geeigneter Entwurfsmuster.
- Die Entwicklung eines Satzes von Benchmarks und Experimenten zur Analyse und Bewertung des Verhaltens von OpenMP Implementierungen.

Als Ganzes liefert diese Arbeit einen methodischen Ansatz zur Entwicklung von paralleler Software. Der Rest dieser Zusammenfassung ist dabei wie folgt strukturiert: Kapitel 2 erläutert die wesentlichen Aspekte von NUMA und stellt die in den Experimenten verwendeten Systeme vor. Kapitel 3 stellt die Entwicklung des Thread Affinity Modells vor, Kapitel 4 präsentiert die Methoden zur Parallelisierung Objekt-orientierter Programme. In Kapitel 5 werden die Ergebnisse anhand einer Fallstudie bewertet. Kapitel 6 schließt mit einer Zusammenfassung sowie einem Ausblick auf zukünftige Forschungsarbeiten.

2 Systemarchitekturen und Benchmarks

Jedes aktuelle Serversystem mit zwei oder mehr Prozessoren ist ein NUMA-System. Das Prinzip des Shared Memory bleibt unberührt, aber der physikalische Speicher wird partitioniert und die Zugriffsgeschwindigkeit hängt damit von der Position des Datums im Speicher relativ zum Rechenkern ab, der den Thread ausführt, welcher den Zugriff tätigt. Abbildung 1 zeigt ein konkretes, hierarchisches NUMA-System.

2.1 Systemarchitekturen

In dieser Arbeit werden exemplarisch zwei Systeme betrachtet, auf denen alle Experimente durchgeführt werden. Das *2-Sockel Intel Westmere System* besteht aus 2 Intel Xeon X5675 Prozessoren mit je 6 Rechenkernen. Es ist repräsentativ für den Standardrechenknoten in aktuellen HPC-Systemen. Die genannten Herausforderungen sind auf einem komplexeren System aber deutlicher zu erkennen und schwieriger zu meistern. Aus diesem Grund wurde als zweites System das *Bull BCS System* ausgewählt. Es besteht aus 4 Boards, die über einen speziellen Interconnect Bull Coherence Switch (BCS) verbunden sind, und somit insgesamt 16 Prozessoren bieten. Das System ist in Abb. 1 dargestellt. Jedes Board ist mit 4 Intel Xeon X7550 Prozessoren mit je 8 Rechenkernen bestückt. Die Abbildung zeigt links die NUMA-Konnektivität innerhalb des Boards, nach der jeder Prozessor mit einem QPI-Link an jeden anderen Prozessor angebunden ist sowie über lokalen Speicher verfügt. Im BCS System werden vier Boards miteinander verbunden, wie rechts dargestellt. Daraus ergibt sich eine NUMA-Hierarchie (auf dem Board und zwischen den Boards) mit unterschiedlichen Zugriffscharakteristiken.

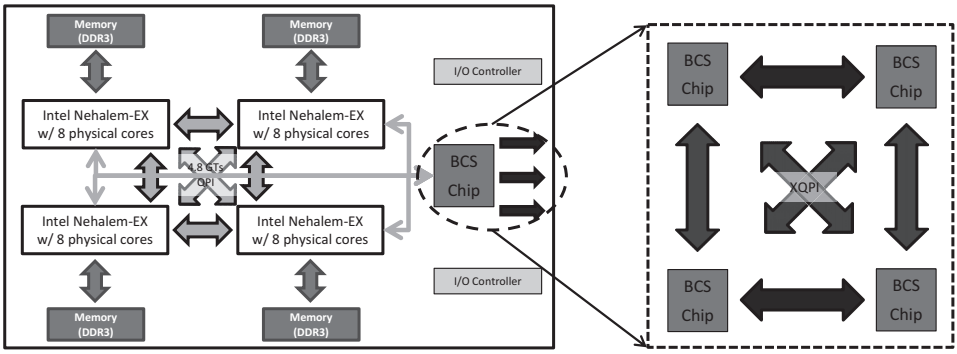

Abb. 1: Architektur des Bull BCS System.

In dieser Zusammenfassung werden Ergebnisse stellvertretend nur für das Bull BCS System betrachtet, da die Herausforderungen in der Programmierung durch die hierarchische Speicherarchitektur eine Obermenge gegenüber einem Standardsystem darstellen. Neuartige Speicher, z.B. mit sehr hoher Bandbreite, werden in Linux ebenfalls als NUMA-Speicherbereiche angeschlossen und sind dementsprechend mit denselben Techniken zu behandeln.

2.2 Benchmarks

Zur Darstellung und Bewertung der Optimierungseffekte der entwickelten Methoden werden drei unterschiedliche Benchmarks verwendet. Diese unterscheiden sich bezüglich des Speicherzugriffsverhaltens und der Lokalität der Datenzugriffe. Zeitliche Lokalität bedeutet, dass kürzlich verwendete Daten mit großer Wahrscheinlichkeit wieder verwendet werden und damit ggfs. noch im Cache vorhanden sind. Räumliche Lokalität bedeutet, dass

nacheinander stattfindende Zugriffe auf Speicherbereiche erfolgen, die im physikalischen Speicher nah beeinander liegen, ggfs. sogar auf derselben Cacheline.

Der STREAM Benchmark misst die tatsächlich erreichbare Speicherbandbreite eines Systems durch die Ausführung von vier unterschiedlichen Vektoroperationen. Diese haben eine hohe räumliche und eine geringe zeitliche Lokalität. Der SpMXV Benchmark realisiert eine Matrix-Vektor-Multiplikation mit einer dünnbesetzten Matrix im CRS-Format (compressed row storage), d.h. ohne Ausnutzung einer bestimmten Struktur der Matrix. Die Leistung von SpMXV ist im Allgemeinen durch die Speicherbandbreite begrenzt. Im iterativen Lösungsverfahren GMRES ist SpMXV die zeitintensivste Operation. Weitere Operationen sind Vektor-Vektor und Vektor-Skalar-Operationen, die in ihrer Charakteristik ähnlich zu den Operationen in STREAM sind.

Zusammengenommen erlauben die drei Benchmarks die Bewertung, ob die angewandten Optimierungsschritte hinsichtlich der Erhöhung der Ausführungsleistung auf komplexen NUMA-Systemen zielführend sind und wann das Optimum erreicht ist. Außerdem stellen sie den Kern vieler technisch-wissenschaftlicher Simulationsanwendungen dar. In dieser Zusammenfassung wird in den folgenden beiden Kapiteln lediglich auf den SpMXV Benchmark eingegangen, da sich damit die Anwendung der entwickelten Methoden exemplarisch darstellen lässt.

3 Softwaredesign für NUMA-Architekturen

Dieses Kapitel motiviert über die Modellierung der Ausführungsleistung die Optimierungsstrategie für NUMA-Architekturen. Darauf aufbauend wird die Entwicklung des Thread Affinity Modells vorgestellt.

Zur Fragestellung der Platzierung von Tasks oder Threads mitsamt Daten in der Systemarchitektur (der sog. *Affinität*), gibt es Vorarbeiten. Über diese hinausgehend führt diese Arbeit das Konzept von sog. *Places* als Abstraktion der Architektur ein. Ein ähnliches Konzept existiert in den Programmiersprachen X10 (ebenfalls als Place [Ch05]) und in Chapel als Locale [CCZ07]. Diese Vorarbeiten haben diese Arbeit beeinflusst, unterscheiden sich als Vertreter der PGAS-Programmiermodelle jedoch in etlichen Aspekten von OpenMP als Shared Memory Programmiermodell und bieten beispielsweise keine vergleichbaren Strategien zur Umsetzung von *Binding*, der Platzierung von Threads in der Systemarchitektur.

3.1 Modellierung der Ausführungsleistung und Systemcharakteristiken

Das Roofline Modell von Williams et al. [WWP09] verknüpft die sog. operationelle Intensität, definiert als Verhältnis von aus dem Hauptspeicher geladenen Daten zu den damit ausgeführten Rechenoperationen, mit der auf einem gegebenen System damit maximal erreichbaren Ausführungsleistung. Die Anwendung auf den SpMXV Benchmark zeigt seine Abhängigkeit von der Speicherbandbreite. Für die als `standard` bezeichnete Matrixgröße

ergibt sich eine operationelle Intensität von 0,1497 FLOPS pro geladenem Byte, was deutlich unter der theoretischen Maximalleistung der Prozessoren liegt.

Die zeilenweise Parallelisierung von SpMXV liefert deutlich weniger Ausführungsleistung als das Modell vorhersagt. Denn neben der Parallelisierung müssen mehrere Faktoren berücksichtigt werden: die Verteilung der Daten in der NUMA-Architektur, die gleichmäßige Verteilung der Rechenaufgaben über die eingesetzten Threads, die Auslegung des Speicherzugriffsmusters auf die Architektur, usw. Alle diese Aspekte müssen in der Entwicklung paralleler Algorithmen und Programme berücksichtigt werden.

3.2 Bewertung der Thread-Affinität

Die Modellierung und die durchgeführten Experimente zeigen grosse Unterschiede in der Ausführungsleistung eines OpenMP Programms auf einer NUMA-Architektur. Als Konsequenz muss die Optimierung vor allem das Speicherzugriffsverhalten betrachten, insbesondere das Verhältnis der Zugriffe auf lokale zu entfernten Daten.

Eine Distanzmatrix zeigt für eine gewählte Metrik die relativen Distanzen zwischen Rechenkernen in der Speicherarchitektur. Für das Bull BCS System ergibt eine solche Darstellung, dass für Zugriffen mit jeweils 8 Threads die Speicherbandbreite zum lokalen (nächesten) Speicher ca. 16 GB/s beträgt, zu benachbartem Speicher auf demselben Board ca. 12 GB/s, und zu entferntem Speicher ca. 4 GB/s.

3.3 Strategien zur Thread-Affinität

Es gibt bereits eine Vielzahl verschiedener NUMA-Systemarchitekturen auf dem Markt und kontinuierlich neue Entwicklungen. Um dennoch Programme entwickeln zu können, die nicht nur für eine gegebene Architektur optimiert sind, enthält das Thread Affinity Modell eine Abstraktion der Maschine sowie Strategien zur Platzierung von Threads, deren Auswahl von den Programmeigenschaften abhängt:

- *close*: Die Threads werden möglichst nahe beieinander platziert, ohne dabei unbedingt auf demselben Kern ausgeführt zu werden. Dabei wird in der Distanzmatrix der Abstand zwischen den ausgewählten Kernen minimiert. Diese Strategie ermöglicht z.B. die schnelle Synchronisation zwischen zwei Threads auf benachbarten Kernen, wenn diese sich einen Cache teilen.
- *spread*: Die Threads werden möglichst weit voneinander entfernt in der Maschinenarchitektur verteilt. Dabei wird in der Distanzmatrix der Abstand zwischen den ausgewählten Kernen maximiert. Diese Strategie ermöglicht z.B. die Optimierung der Speicherbandbreite durch Nutzung mehrerer Speichercontroller bereits ab einer geringen Threadanzahl.
- *master*: Die Threads werden möglichst auf demselben Kern ausgeführt, d.h. der Abstand in der Distanzmatrix ist 0. Diese Strategie ist ein Spezialfall und beschleunigt beispielsweise gewisse I/O-Operationen.

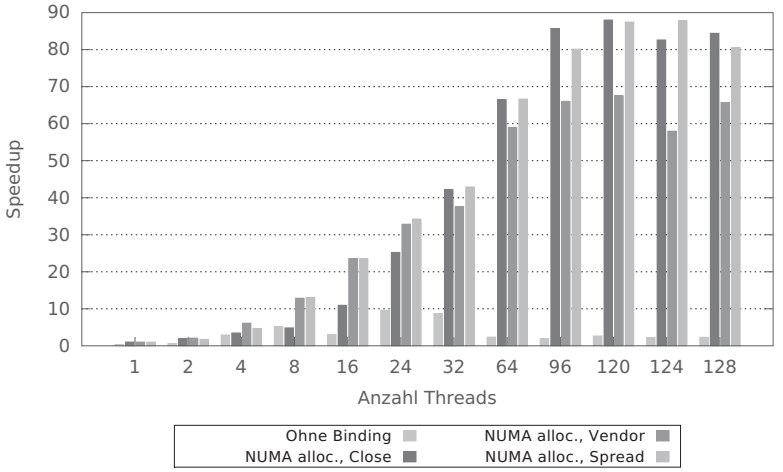

Abb. 2: SpMXV auf dem Bull BCS System mit Thread-Affinität.

Das Konzept der Places liefert eine Abstraktion der Hardware: ein sog. *Place* stellt eine nicht leere Menge von Ausführungseinheiten dar, in der ein Thread ausgeführt wird, aus der er jedoch nicht ausweichen kann. Eine gegebene Maschine wird dabei für ein OpenMP-Programm als eine geordnete Liste der Places (die sog. *Place List*) dargestellt [Ei12]. Diese wird von der Laufzeitumgebung bereitgestellt. Der Benutzer des OpenMP-Programms kann dabei die Konstruktion der Liste über eine Umgebungsvariable festlegen: ein abstrakter Namen steuert, ob die Thread-Affinität z.B. auf Ebene der logischen oder der physikalischen Kerne oder auf Ebene der Sockel erfolgen soll. Letzteres ist ein wichtiger Anwendungsfall in der hybriden Parallelisierung. Zur Konstruktion der Place List und die Umsetzung der Thread-Affinität zur Laufzeit wird in [Te16] ein effizienter Ansatz vorgestellt, der auch die Schachtelung von Parallelität auf mehreren Ebenen (z.B. in Bibliotheken) unterstützt.

Abbildung 2 zeigt das Ergebnis der Anwendung der Affinity Strategien im SpMXV Benchmark auf dem Bull BCS System. Ohne jegliche Affinität kann nur ein sehr geringer Speedup erreicht werden (■). Der Unterschied zwischen den Strategien wird vor allem bei kleineren Threadzahlen deutlich, bei denen *spread* (■) eine höhere Speichbandbreite für SpMXV erzielt. Dazu kommt, dass die Laufzeiten der parallelen Programme nun keinen zufälligen Schwankungen mehr unterliegen. Die vorgestellte Lösung ist herstellerspezifischen Lösungen (■) überlegen.

3.4 Memory-Affinität

Für die Messergebnisse aus Abb. 2 wurde bereits eine NUMA-Optimierung realisiert: für SpMXV wurde die Matrix so über die NUMA-Knoten verteilt, wie in der nachfolgenden Berechnung darauf zugegriffen wird [Te08a].

Eine Besonderheit bei Task-parallelen Programmen ist, dass in diesem Konzept dem Programmierer keine Möglichkeit gegeben wird zu beeinflussen, durch welche Threads und damit wo im System die Tasks abgearbeitet werden. Durch die Durchführung geeigneter Experimente kann festgestellt werden, dass die aktuellen OpenMP-Implementierungen dabei keine NUMA-Optimierung durchführen und entsprechend die Ausführungsleistung hinter den Möglichkeiten zurückbleibt. Um dem zu begegnen, können zwei Programmiermuster eingesetzt werden, mit deren Hilfe die OpenMP Laufzeitumgebung bei der Ausführung unterstützt werden kann: *single-producer* und *parallel-producer* [Te12]. Bei letzterem werden die Tasks in geeigneter Weise parallel erzeugt und mit großer Wahrscheinlichkeit nachfolgend von Threads abgearbeitet, welche eine passende Daten-Affinität besitzen.

Neben den in dieser Zusammenfassung vorgestellten Ergebnissen werden in [Te16] Vorschläge gemacht, wie direkt auf die Platzierung von Daten oder die Migration von Daten mittels Sprachkonstrukten eingewirkt werden kann. Eine Weiterentwicklung dieser Vorschläge wird derzeit zur Aufnahme in OpenMP diskutiert. Ebenfalls wird untersucht, in wie weit ein Ansatz zur Erhöhung der Daten-Affinität innerhalb des Betriebssystems realisiert werden kann.

4 Parallelisierung mit Objekt-orientierten Abstraktionen

Dieses Kapitel führt den Nachweis, dass die Konzepte des Affinity Modells in Objekt-orientierten C++-Programmen ohne Einebnung der Klassenhierarchie eingesetzt werden können.

Blitz++ [Ve98] hat den Einsatz von Expression Templates in C++ entwickelt um die Übersetzung von komplexen Ausdrücken in effizienten Code zu ermöglichen. Gamma et al. [Ga95] haben die Formulierung von allgemeingültigen Entwurfsmuster eingeführt.

4.1 Abstraktionen für Thread- and Task-Parallelität

Die Parallelisierung von C++-Programmen mittels OpenMP Pragmas kann dazu führen, dass dem Compiler die Anwendung von Optimierungen wie Expression Templates unmöglich wird. Dies liegt daran, dass aus Sicht des Compilers die Pragmas nicht bekannte Seiteneffekte induzieren können.

Bei der Parallelisierung eines bestehenden Objekt-orientierten Programmcodes gibt es mehrere Wahlmöglichkeiten, wo die Parallelisierung angesetzt werden kann. Am Beispiel von numerischen Lösern werden in [Te16] verschiedene Ansätze untersucht: generell ist es möglich, innerhalb der einzelnen Funktionen eines Objektes die Parallelisierung unterzubringen, jedoch entsteht dabei in jedem Aufruf ein Overhead der Parallelisierung; die Platzierung außerhalb spart diesen Overhead, führt jedoch häufig zu größeren Änderungen am Programmcode.

Als bester Kompromiss erweist sich der Einsatz von Expression Templates bei der Parallelisierung des Zuweisungsoperators (`operator=`), um komplexe Ausdrücke in einer einzelnen OpenMP Region zu parallelisieren und so deutlich Overhead einzusparen. Die Optimierung durch den Compiler bleibt dabei weiterhin möglich. Mittels des Strategy Muster kann dabei zwischen Details der Implementierung ausgewählt werden. Templates in Verbindung mit dem Adapter Muster erlauben es, die parallelisierten Objekte in Anwendungen einzubinden. Dies ist unter anderem für die Anwendung DROPS erfolgreich [Te08b].

4.2 Abstraktionen für NUMA

Um das Affinity Modell in C++-Programmen einzusetzen, ist die Unterstützung der Trennung von Allokation und Initialisierung mittels Allokatoren notwendig. In [Te16] wird gezeigt, wie entsprechende Allokatoren zu implementieren sind und, dass diese die Anforderungen erfüllen. Damit kann die Objekt-orientierte Art der Programmierung die Parallelisierung unterstützen um die Ausführungsleistung zu steigern. Bei SpMXV und GMRES ist dies beispielsweise der Fall, da die Logik der Datenzugriffsoptimierung in die Objekte integriert wurde und damit Zugriff auf private Informationen wie die Matrixstruktur erhält.

Im Rahmen der Entwicklung an mehreren Anwendungen, unter anderem DROPS, wurde offensichtlich, dass der OpenMP Standard in Versionen vor 3.0 keine geeignete Unterstützung für C++ enthält. Beispielsweise war nicht definiert, zu welchem Zeitpunkt zusätzliche Instanzen eines privatisierten Objekts erstellt werden, und in welcher Anzahl. Um dies zu klären wurden semantische Vorschläge gemacht, die in den Standard OpenMP 3.0 eingeflossen sind.

5 Anwendungsfallstudien

Anhand von drei Anwendungen sowie zwei weiteren Beispielen wird in [Te16] gezeigt, dass die vorgestellten Konzepte in den OpenMP Standard integriert und von aktuellen Compilern implementiert wurden. Daraus resultiert eine deutliche Steigerung der Ausführungsleistung in Anwendungen, wie am Beispiel FIRE hier gezeigt wird.

FIRE (Flexible Image Retrieval Engine, am Lehrstuhl I6 der RWTH Aachen entwickelt, C++) liefert zu einem Anfragebild die k ähnlichsten Bilder einer Datenbank zurück, indem mehrere Bildeigenschaften verglichen werden. Hier kann sowohl eine Parallelisierung über die Anfragen (omp1 in Abb. 3) und die Bilder in der Datenbank (omp2) eingesetzt werden, als auch ein Task-basierter Ansatz (tasks), wobei letzterer die beste Leistung liefert. Abb. 3 zeigt den erreichten Speedup auf dem Bull BCS System, der durchweg nahe am Optimum (linear) von 144 Threads liegt.

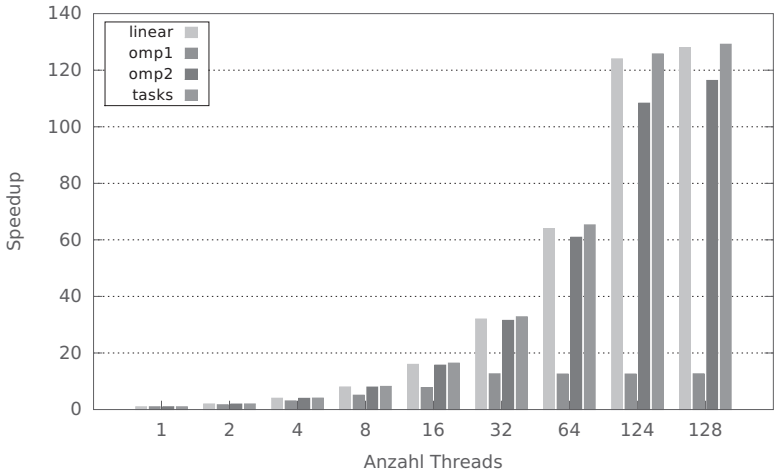

Abb. 3: Vergleich des Speedup von FIRE auf dem Bull BCS System.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurden Abstraktionen zur effizienten Programmierung von Multi-Core Architekturen mit hierarchischem Speicher vorgestellt. Dabei wurde zunächst das Affinity Modell von OpenMP 4.0 entwickelt und anschließend gezeigt, mit welchen Konzepten dies auch in C++-Programmen umgesetzt werden kann. Das Einfließen in den Programmierstandard OpenMP, der von allen wesentlichen Compilerherstellern implementiert wird, macht die Ergebnisse dieser Arbeit der HPC-Gemeinde zugänglich.

Die fortschreitende Entwicklung der Hardware erfordert eine kontinuierliche Anpassung der Programmiermodelle. Dies gilt insbesondere für den Bereich des Hochleistungsrechnens, in dem die aktuellste Hardware eingesetzt und die höchste Leistung gefordert wird. Meine aktuellen Arbeiten zielen auf die Einbringung der entwickelten Konzepte für die Task-parallele Programmierung in den Standard OpenMP ab. Dabei wird berücksichtigt, dass moderne Systeme ein vermehrt dynamisches Verhalten in der Ausführungsleistung zeigen und Programmiersysteme damit umgehen müssen.

Literaturverzeichnis

[As09] Asanovic, K.; Bodik, R.; Demmel, J.; Keaveny, T.; Keutzer, K.; Kubiatowicz, J.; Morgan, N.; Patterson, D.; Sen, K.; Wawrzynek, J.; Wessel, D.; Yelick, K.: A View of the Parallel Computing Landscape. Commun. ACM 52/10, S. 56–67, Okt. 2009.

[CCZ07] Chamberlain, B.; Callahan, D.; Zima, H.: Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications 21/3, S. 291–312, 2007.

[Ch05] Charles, P.; Grothoff, C.; Saraswat, V.; Donawa, C.; Kielstra, A.; Ebcioglu, K.; von Praun, C.; Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '05, ACM, San Diego, CA, USA, S. 519–538, 2005.

[Ei12] Eichenberger, A. E.; Terboven, C.; Wong, M.; an Mey, D.: The Design of OpenMP Thread Affinity. In: OpenMP in a Heterogeneous World. Bd. 7312, Lecture Notes in Computer Science, Springer, S. 15–28, 2012, ISBN: 978-3-642-30960-1.

[Ga95] Gamma, E.; Helm, R.; Johnson, R. E.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Te08a] Terboven, C.; an Mey, D.; Schmidl, D.; Jin, H.; Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: MAW '08: Proceedings of the 2008 Workshop on Memory access on Future Processors: a solved problem? ACM, ACM, Ischia, Italy, S. 377–384, 2008, ISBN: 978-1-60558-091-3.

[Te08b] Terboven, C.; Spiegel, A.; an Mey, D.; Gross, S.; Reichelt, V.: Experiences with the OpenMP Parallelization of DROPS, a Navier-Stokes Solver Written in C++. In: OpenMP Shared Memory Parallel Programming. Bd. 4315, Lecture Notes in Computer Science, Springer, S. 95–106, 2008.

[Te12] Terboven, C.; Schmidl, D.; Cramer, T.; an Mey, D.: Task-Parallel Programming on NUMA Architectures. Euro-Par 2012 Parallel Processing/, S. 638–649, 2012.

[Te16] Terboven, C.: Abstractions for Performance Programming on Multi-Core Architectures with Hierarchical Memory, Diss., RWTH Aachen University, 2016.

[Ve98] Veldhuizen, T. L.: Arrays in Blitz++. In: ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments. Springer-Verlag, London, UK, S. 223–230, 1998.

[WWP09] Williams, S.; Waterman, A.; Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52/4, S. 65–76, 2009, ISSN: 0001-0782.

Christian Terboven wurde am 22. Februar 1980 in Kamp-Lintfort geboren. Seit seinem Studium der Informatik arbeitet er am IT Center und Lehrstuhl für Hochleistungsrechnen der RWTH Aachen University. Dort hat er in 2015 die Leitung der HPC Gruppe übernommen. Seit 2006 ist Christian Terboven Mitglied im OpenMP Language Committee, dem Gremium zur Weiterentwicklung des Sprachstandards. Er leitet dort die Affinity Teilgruppe.