University of Applied Sciences
Georg-Simon-Ohm Fachhochschule Nürnberg
Fachbereich Informatik

Diplomarbeit

Freie Wissenschaftliche Arbeit zur Erlangung des akademischen Grades
Diplom-Informatiker (FH) an der Georg-Simon-Ohm Fachhochschule Nürnberg

C++-Programmiertechniken für High Performance Computing auf Systemen mit nichteinheitlichem Speicherzugriff unter Verwendung von OpenMP

Betreuender Hochschullehrer:	Prof. Dr. Reinhard Eck
Betreuer RRZE:	Dr. Georg Hager
Autor:	Holger Stengel
Matrikelnummer:	800690

Eingereicht am:	28. September 2007

Zusammenfassung

Shared-Memory-Multiprozessorsysteme mit nichteinheitlichem Speicherzugriff, wozu insbesondere ccNUMA-Rechner (ccNUMA = cache-coherent Non-Uniform Memory Access), aber auch manche Multi-Core-Systeme gehören, sind im High Performance Computing (HPC) weit verbreitet, sowohl als Bausteine in Clustern als auch in Supercomputer-Installationen wie dem HLRB II am LRZ München. Beim parallelen Programmieren mittels OpenMP ist dabei auf eine weit gehende Lokalität beim Speicherzugriff zu achten, damit das Netzwerk nicht durch nichtlokalen Datentransfer zum Flaschenhals für die Performance der Applikation wird. In traditionellen Programmiersprachen wie C und Fortran ist dieses Problem relativ leicht lösbar. Mit C++, das im HPC immer mehr an Bedeutung gewinnt, stößt man jedoch auf prinzipielle Probleme, die in den vielfältigen Automatismen des objektorientierten Programmierens begründet liegen.

Die Arbeit hat zum Ziel, Richtlinien und Programmiertechniken für OpenMP unter C++ zu entwickeln, die auf Systemen mit nichteinheitlichem Speicherzugriff zu effizienter, d.h. performanter und skalierbarer paralleler Software führen. Ausgegangen wird von einer grundlegenden Analyse der existierenden Lokalitätsprobleme und ihrer Ursachen. Darauf aufbauend werden exemplarisch bestimmte Programmiertechniken auf ihre mögliche Optimierbarkeit bei einfachen Schleifenbenchmarks untersucht. Containerklassen nehmen dabei eine Schlüsselrolle ein. Schließlich wird die Leistungsfähigkeit der entwickelten Methoden anhand von Benchmarktests überprüft.

Inhaltsverzeichnis

1 Grundlagen

1.1 Struktur der Arbeit

Die vorliegende Arbeit beschäftigt sich mit Optimierungen von C++-Codes im High Performance Computing (HPC) unter dem besonderen Gesichtspunkt von Systemen mit gemeinsamem Speicher und nichteinheitlichem Speicherzugriff. Im ersten Kapitel werden nach einer grundsätzlichen Einführung in das Thema HPC und einer Erörterung der Speicherhierarchien in modernen Mikroprozessorsystemen zunächst die verschiedenen Programmiermodelle für paralleles Rechnen vorgestellt. Im Bereich des Shared-Memory unterscheidet man zwischen UMA- und NUMA-Architekturen, welche ausführlich erörtert werden. Die für die Ermittlung von Performancezahlen verwendeten Benchmarks und ihre Implementierung sind von zentraler Bedeutung für die spätere Argumentation und finden deshalb genauso Erwähnung wie die prinzipiellen Systemarchitekturen der untersuchten Rechnersysteme.

Das zweite Kapitel widmet sich den zentralen Hindernissen beim Programmieren auf ccNUMA-Systemen, dem Lokalitätsproblem und dem Bandbreitenproblem. Anhand einfacher Benchmarks wird der grundsätzliche Lösungsweg des Page Placements aufgezeigt, der in allen HPC-relevanten Programmiersprachen einsetzbar ist. Schließlich wird auf den besonderen und oft vernachlässigten Problemkreis des Process Pinnings eingegangen, der nicht nur bei ccNUMA-, sondern seit Einführung der Multicore-Prozessoren auch bei UMA-Archtekturen eine große Rolle spielt.

Im dritten Kapitel wird der Einsatz von C++ im High Performance Computing und die damit verbundenen Hindernisse auf ccNUMA-Systemen erörtert. Es wird gezeigt, dass die im zweiten Kapitel umrissenen Lösungen bei Verwendung von Standard-Programmiertechniken unter C++ oft fehlschlagen, was in den üblicherweise praktischen, hier jedoch hinderlichen Automatismen der Sprache begründet liegt. Insbesondere sind die allgegenwärtigen STL-Containerklassen für die parallele Programmierung, geschweige denn unter Berücksichtigung von Lokalitätsbedingungen, nicht ausgelegt. Dies wird ausführlich anhand der `vector<>`-Klasse aus der Standardbibliothek erörtert. Als mögliche Auswege werden die Verwendung von NUMA-fähigen Allokatorklassen und ein neu programmierter NUMA-Container vorgeschlagen, der mit den STL-Algorithmen kompatibel ist und eine genaue Kontrolle des Page Placements erlaubt. Abschließend werden natürlicherweise segmentierte Datenstrukturen studiert, die in Form spezieller Containerklassen STL-kompatibel implementiert werden und mit Hilfe sogenannter Merkmalsklassen (Traits) das Design von Algorithmen erlauben, die direkt auf das segmentierte und dadurch perfekt auf ccNUMA- und Multicore-Architekturen angepasste Datenlayout reagieren können.

Das vierte Kapitel widmet sich der Anwendung aller dargestellten Lösungsmöglichkeiten der NUMA-Problematik im Rahmen eines anwendungsnahen Gauß-Seidel-Benchmarks, zieht ausführliche Performancevergleiche und diskutiert kritisch die verschiedenen Anwendungsbereiche der verwendeten Methoden.

Das fünfte und letzte Kapitel fasst schließlich die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche weiterführende Projekte.

1.2 HPC

Als *High Performance Computing* bezeichnet man die Lösung wissenschaftlicher Probleme mit numerischen Mitteln unter Einsatz massiver Rechnerressourcen. Die numerische Simulation mit HPC als ihrem Werkzeug hat sich mittlerweile als drittes Standbein neben Theorie und Experiment fest etabliert, nicht nur im akademischen, sondern auch im industriellen Umfeld [1]. Angesichts beachtlicher Invesitionen in Hard- und Software nimmt dabei die optimierte Nutzung der zur Verfügung stehenden Ressourcen einen wichtigen Platz ein. Da es sich zumeist (aber nicht immer) um *parallele* Programme handelt, also Codes, die möglichst viele Recheneinheiten gleichzeitig nutzen, ist die Geschwindigkeitsoptimierung eine überaus komplexe Aufgabe und setzt detaillierte Kenntnisse über Hardware und Programmiertechniken voraus, die normalerweise nur in Spezialistenteams wie der HPC-Gruppe des Regionalen Rechenzentrums Erlangen (RRZE) [2] vorhanden sind. Zentrale Fragen sind z. B.:

- Was ist der limitierende Faktor für die Rechengeschwindigkeit (Performance) eines Programmes?
- Welche Codeteile verbrauchen die meiste Rechenzeit und warum?
- Kann eine Programmänderung an geeigneter Stelle eine Verbesserung der Geschwindigkeit herbeiführen?
- Wird das Programm, wenn es auf N Recheneinheiten (Prozessoren) „schnell“ läuft, auf $M > N$ Prozessoren M/N mal schneller laufen? Was begrenzt diese Skalierbarkeit?

Der Wissenschaftler oder Ingenieur als Anwender oder sogar Entwickler der Software kann sich aus Zeitgründen normalerweise nicht genügend weit in die Materie einarbeiten, um ein volles Verständnis aller Performanceaspekte zu entwickeln. Er benötigt einfache „Kochrezepte“, die *im Allgemeinen* zu guter Performance führen.

Eines der Ziele dieser Arbeit ist die Anfertigung solcher „Rezepte“ speziell für die Sprache C++, die im wissenschaftlichen Bereich große Bedeutung gewonnen hat. Der Grund für diese Entwicklung ist, dass die fortschreitende Komplexität der Softwaresysteme nur noch durch objektorientierte Methoden beherrscht werden kann. Der ständig wieder aufflammenden Diskussion um die Eignung von C++ als Programmiersprache im High Performance Computing sollte man dabei pragmatisch begegnen: Bei Beachtung einiger wohlbekannter Regeln [3, 4] lässt sich C++ sehrwohl als performante Sprache einsetzen, und bei wirklich „maschinennahen“ Codeteilen ist immer auch die Verwendung von C, Fortran oder Assembler möglich. In der Tat geht wirklich effizienter, komplexer C++-Code heute genau diesen Weg [5].

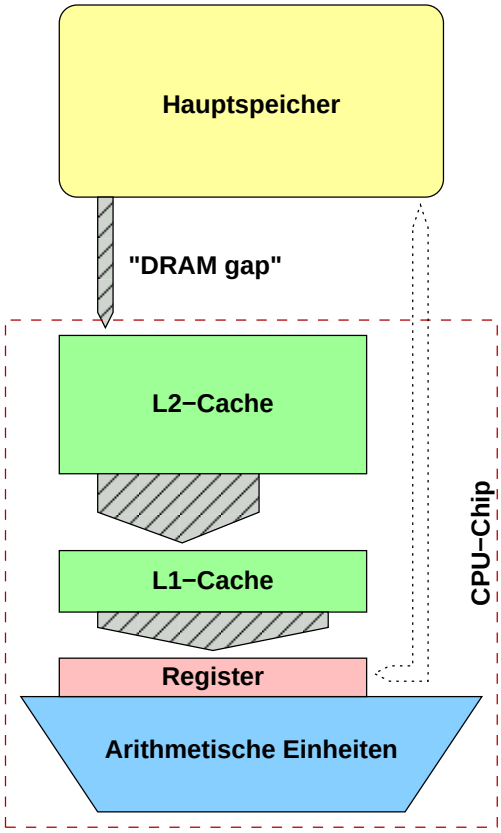

Abbildung 1.1: Modell der Speicherhierarchie in einem cachebasierten Mikroprozessor. Direkter Zugriff von den Prozessorregistern auf den Hauptspeicher ist nicht bei allen Architekturen möglich. (nach [6])

1.3 Speicherarchitektur moderner Mikroprozessoren

Die Ausführungsgeschwindigkeit von (HPC-)Applikationen wird häufig von Übertragungsrate und Latenz der Speicheranbindung limitiert. Neben der Übertragungsrate (hierfür hat sich der Begriff „Bandbreite“ etabliert) ist die Latenz die zweite wichtige Metrik zur Performancecharakterisierung von Netzwerken. Sie beschreibt die Verzögerungszeit, die durch Kommunikationsverwaltung wie z.B. den Auf- und Abbau einer Verbindung entsteht. Dieser Engpaß führt dazu, dass die Recheneinheiten der CPU nicht voll ausgelastet werden, da die für die Rechnung erforderlichen Daten nicht schnell genug bereitgestellt werden können. Der Grund hierfür ist ein als „DRAM-gap“ bekanntes Phänomen, wonach Prozessorgeschwindigkeiten mit fortschreitender technologischer Entwicklung schneller wachsen als die von Hauptspeichermodulen. Um diese Kluft zu verbergen, wird ein im Vergleich zum Hauptspeicher kleiner und schneller Zwischenspeicher zwischen Prozessor und Hauptspeicher platziert. Dieser *Cache*,der heute in den meisten Fällen direkt auf dem CPU-Chip zu finden ist, ist üblicherweise in zwei oder drei Ebenen aufgeteilt, da Caches aus Kostengründen nicht gleichzeitig hohe Kapazität und hohe Datenübertragungsrate bzw. niedrige Latenz haben können. Je näher der Cache also am Prozessor ist, desto schneller und kleiner ist er.

Damit eine arithmetische Einheit ein Datum in einem Register zur Verfügung hat, muss es im Allgemeinen aus dem Hauptspeicher geladen und durch diese *Cachehierarchie* zum Prozessor weitergereicht werden (siehe Abbildung 1.1). Ein Geschwindigkeitsvorteil ergibt sich, falls das benötigte Datum bereits in der innersten Cacheebene vorhanden ist und von dort geladen werden kann. In diesem Fall spricht man von einem *Cache-Hit*. Bei einem *Cache-Miss* muss

das Datum aus einer äußeren Ebene oder dem Hauptspeicher geholt werden. Das Datum wird für eventuelle weitere Zugriffe im Cache abgelegt. Bei voll belegtem Cache wird hierzu von der Hardware ein älteres Element entfernt. Eine Applikation profitiert vom Cache, wenn ihr Datenzugriffsmuster zeitliche Lokalität aufweist. Das Datum wird also wiederholt geladen und verarbeitet, bevor es aus dem Cache verdrängt wird.

Das Zugriffsmuster vieler Anwendungen ist jedoch dominiert von Datenströmen, die einmalig gelesen, verarbeitet und geschrieben werden. Eine Wiederverwendung von Cache-Elementen findet also nicht statt. Hier ergibt sich stattdessen ein Vorteil aus der zeilenbasierten Cacheorganisation. Bei einem Cache-Miss wird nicht nur das eine angeforderte Datum aus dem Hauptspeicher übertragen, sondern eine sogenannte *Cachezeile*, die aus mehreren, hintereinander im Speicher liegenden Datenelementen besteht. Benachbarte Elemente können dann direkt aus dem Cache gelesen werden. Dies erhöht die Rate von Cache-Hits bei Programmen mit räumlich lokalem Zugriffsmuster. Bei 64 Byte Cachezeilenlänge und einem Array aus doppelt genauen Fließkommawerten (8 Byte) treten bei kontinuierlichem Zugriff also nur ein Achtel der Cache-Misses auf.

1.4 Parallele Programmiermodelle

Unter einem *Programmiermodell* versteht man die Sicht des Programmierers auf das Rechnersystem, konkret z.B. die Organisation des Hauptspeichers. Das Programmiermodell ist eine Abstraktionsschicht über der tatsächlich zugrundeliegenden Hardware. Zur Entwicklung paralleler Algorithmen haben sich zwei Modelle herausgebildet, die sich hauptsächlich in der Art der Kommunikation unterscheiden.

Zuerst wird die Variante mit *verteiltem Speicher* kurz vorgestellt, danach die für diese Arbeit grundlegende Version mit *gemeinsamem Speicher*. Im Text werden synonym die üblichen englischen Bezeichnungen *Distributed-Memory* bzw. *Shared-Memory* verwendet.

1.4.1 Verteilter Speicher

Beim Distributed-Memory-Programmiermodell besteht ein paralleles Programm aus mehreren Prozessen, von denen jeder exklusiven Zugriff auf einen Teil des gesamten Hauptspeichers hat. Kein Prozess kann also direkt auf den Speicher anderer Prozesse zugreifen. Über ein Kommunikationsnetzwerk können die Prozesse Nachrichten austauschen, z.B. zur Steuerung der Parallelverarbeitung oder um Zwischenergebnisse zu verteilen. Für den Nachrichtenaustausch hat sich der MPI (Message Passing Interface) [7, 8] Standard etabliert. Auf allen beteiligten Prozessen läuft das in einer der üblichen Programmiersprachen (C, C++, Fortran) geschriebene Programm jeweils seriell ab. Jeder Prozess ist durch eine eindeutige Nummer, den „MPI-Rang“ identifizierbar. Der Programmierer muss mit entsprechenden Funktionen der MPI-Bibliothek den Nachrichtenaustausch implementieren. Außerdem muss die Verteilung der Arbeitspakete auf die Prozesse, beispielsweise im Rahmen einer Gebietszerlegung, von Hand geschehen. Unter Gebietszerlegung versteht man das Zerschneiden des Problems in

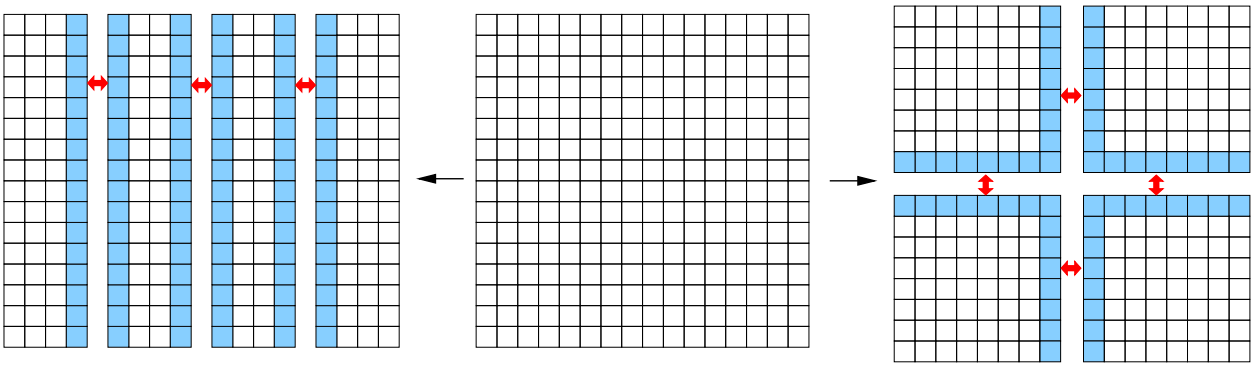

Abbildung 1.2: Mögliche Zerteilungen eines Gebietes für vier Prozessoren bei einem „nächster Nachbar“-Kommunikationsmuster. Der Kommunikationsaufwand ist im rechten Fall geringer, da weniger Elemente ausgetauscht werden müssen (schattierte Felder). (aus [6])

kleinere Bereiche, die dann jeweils von einem Prozess bearbeitet werden. Bei vielen Anwendungen (z.B. Gauß-Seidel, siehe S. 13) bestehen Abhängigkeiten zwischen Nachbarpunkten, d.h. während des Programmlaufes müssen die Prozesse kommunizieren. Das Kommunikationsnetzwerk ist üblicherweise im Vergleich zur Ausführungsgeschwindigkeit sehr langsam und latenzbehaftet. Deshalb sollte sowohl das Kommunikationsvolumen als auch die Kommunikationshäufigkeit möglichst gering gehalten werden. Dieses kann durch die Art der Gebietszerlegung beeinflusst werden (siehe Abbildung 1.2). Hier ist auf minimale Schnittlängen und symmetrische Aufteilung des Gebietes zu achten. Die nachrichtengekoppelte Parallelisierung mit MPI ist sehr flexibel und effizient, jedoch im Vergleich zu dem im Folgenden beschriebenen Ansatz mit deutlich höherem Entwicklungsaufwand verbunden.

1.4.2 Gemeinsamer Speicher

Shared-Memory-Programmierung erfordert keinen expliziten Nachrichtenaustausch. Kommunikation erfolgt hier über gemeinsame Variablen, also Speicherbereiche. OpenMP (Open Multi Processing) hat sich zum Standard für diese Art der Parallelisierung entwickelt. Im Gegensatz zu MPI werden bei OpenMP nicht mehrere Prozesse für die Parallelverarbeitung erzeugt, sondern mehrere Threads innerhalb eines einzigen Prozesses. Während ein Prozess stets einen eigenen Adressraum hat, nutzen alle Threads eines Prozesses den selben Adressraum, was jedem Thread direkten Zugriff auf alle Daten erlaubt. OpenMP bietet dem Benutzer einen Satz von Compilerdirektiven und einige Bibliotheksfunktionen für die Programmiersprachen C/C++ und Fortran. Diese Spracherweiterung wird von allen aktuellen Compilern unterstützt, so dass keine weitere Software erforderlich ist. Falls OpenMP vom Compiler nicht unterstützt wird oder die Unterstützung deaktiviert ist, werden die die Direktiven als Kommentare interpretiert. So kann eine vollständig serielle Version des Programms erzeugt werden. Umgekehrt kann ein bestehender serieller Code inkrementell parallelisiert werden, indem Schritt für Schritt Compilerdirektiven eingefügt werden. OpenMP unterstützt in besonderem Maße die Parallelisierung von Schleifen, was numerischen Anwendungen, die üblicherweise schleifendominiert sind, entgegenkommt.

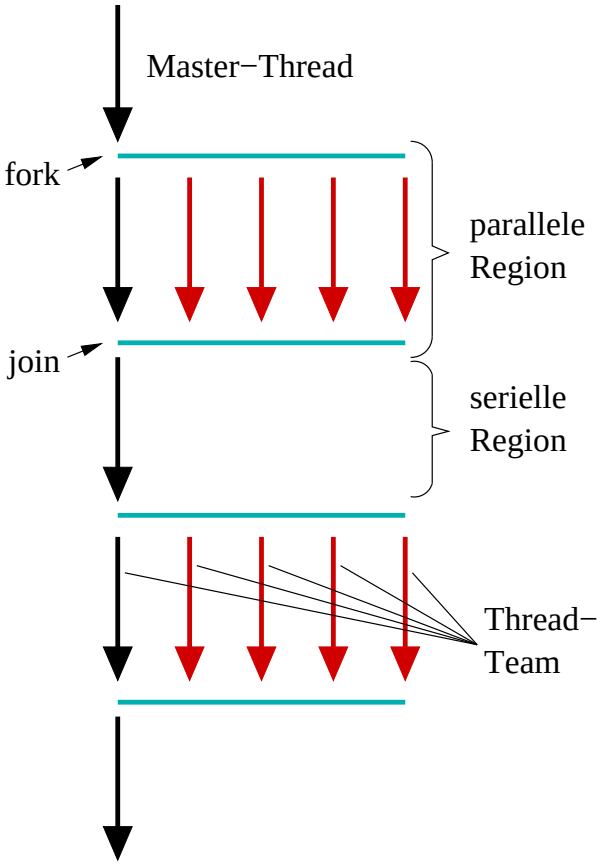

Abbildung 1.3: OpenMP Thread-Operationen: Der Master-Thread spaltet sich in einer parallelen Region zu einem Thread-Team auf (fork). Danach werden die Threads bis zur nächsten parallelen Region zusammengeführt (join). (nach [6])

1.4.3 Distributed-Shared-Memory

Bei Shared-Memory-Systemen stellt die vergleichsweise einfache Programmierung und Handhabung einen Vorteil gegenüber Distributed-Memory-Systemen dar. Diese wiederum sind, z.B. in Form von PC-Clustern, weit verbreitet und kostengünstig. Von den üblichen parallelen Programmiermodellen (siehe Kapitel 1.4.1 und 1.4.2) kann hier prinzipbedingt nur MPI zum Einsatz kommen. Diese Lücke wird durch sogenannte *Distributed-Shared-Memory* (DSM) geschlossen. Hierbei handelt es sich um simulierte Shared-Memory-Systeme, in denen eine Softwareschicht die Abstraktion der (Distributed-Memory-) Hardware übernimmt. Damit kann z.B. bestehender, mit OpenMP parallelisierter Code auch auf Distributed-Memory-Rechnern ausgeführt werden.

1.5 Klassifizierung von Rechnerarchitekturen mit gemeinsamem Speicher

In Shared-Memory-Rechnersystemen wird der gesamte physikalische Speicher in einen einzigen logischen Speicheradressraum abgebildet. Auf diesen haben alle Prozessoren gleichberechtigten Zugriff. Man unterscheidet zwei Arten von Shared-Memory-Rechnern an ihrem physikalischen Speicherlayout (Performance-Charakteristik). Bei *UMA*-Systemen (Uniform Memory Access) ist der Zugriff auf ein beliebiges Speicherwort für jeden Prozessor gleich

schnell/teuer. Sie werden deshalb auch als symmetrische Multiprozessoren (SMP) bezeichnet. Im Gegensatz hierzu stehen *NUMA*-Architekturen (Non Uniform Memory Access), bei denen die Kosten für den Zugriff auf ein Datum von dessen Lage im Speicher abhängen. Handelsübliche Dualcore-PCs gehören zur UMA-Klasse. Im Gegensatz dazu verwenden große Shared-Memory-Systeme wie die SGI Altix Supercomputer eine NUMA-Architektur.

1.5.1 UMA

Der Zugriff auf die geteilte Ressource Hauptspeicher wird in einem UMA-System von einer zentralen Komponente verwaltet (die zum Beispiel die Caches der einzelnen CPUs kohärent hält). Im einfachsten Fall sind dies die Businterfaces der einzelnen Kerne oder ein Chipsatz, an den diese angeschlossen sind. Über den Bus kann zu einem Zeitpunkt genau ein Prozessor mit dem Speicher kommunizieren. Besser ist ein Switch, der allen Prozessoren gleichzeitig Zugriff auf den Hauptspeicher ermöglicht. Aus Kostengründen wird diese Variante oft in abgeschwächter Form, d.h. mit reduzierter aggregierter Bandbreite implementiert. Die zentrale UMA-Eigenschaft der einheitlichen Speicherzugriffskosten wird durch den Einsatz von Mehrkernprozessoren modifiziert (Pseudo-UMA). Am Beispiel eines Knotens des Woodcrest-Clusters am RRZE kann dies verdeutlicht werden. In diesem System sind zwei Dualcore-Prozessoren jeweils über einen eigenen Bus an der Speicherverwaltungseinheit angeschlossen (Abbildung 1.4). Die aggregierte Bandbreite dieser beiden Verbindungen (Frontside Busse, FSB) ist in der Praxis nicht ganz doppelt so groß wie die eines einzelnen Prozessors auf dem Sockel. Die beiden Kerne innerhalb eines Sockels müssen sich dagegen die Bandbreite eines einzigen FSB teilen. Benötigt ein paralleles Programm mit zwei Threads viel Speicherbandbreite, so ist es ratsam, auf jedem der beiden Sockel einen Thread zu platzieren. Auf diese Weise kann jeder Thread fast die volle Bandbreite „seines“ FSBs nutzen. Andererseits verfügen die beiden Kerne eines Woodcrest-Prozessors über einen gemeinsamen Level 2 Cache. Kann das Datenzugriffsmuster der Applikation diesen Nutzen, sollten beide Threads auf einen Sockel gelegt werden. (Die hierzu benötigte Technik des Pinning wird in Kapitel 2.2 beschrieben.) Besteht also ein Flaschenhals in der Speicheranbindung, so ist auch bei vermeintlichen UMA-Systemen eine gezielte Speichernutzung lohnend. Die Verwendung von vier statt zwei Threads bringt hier keinen signifikanten Vorteil.

1.5.2 NUMA

Der gemeinsame Hauptspeicher ist bei NUMA-Systemen physikalisch auf die Prozessoren verteilt, so dass es für jeden Prozessor lokalen und entfernten Speicher gibt. Letzterer ist transparent über ein Netzwerk angebunden. Ein NUMA-Rechner besteht aus mindestens zwei *Lokalitätsdomänen* (LD). Eine Lokalitätsdomäne beinhaltet die lokal an einen Speicherbereich angeschlossenen Prozessoren. Nichtlokale Speicherzugriffe, also Zugriffe auf Speicher in anderen Domänen, sind, wie oben beschrieben, generell langsamer als lokale Zugriffe (siehe Kapitel 2 für Details).

SGI Altix Systeme sind typische NUMA-Implementierungen, wie sie im HPC-Umfeld zum Einsatz kommen. Als Beispiel sei hier der Bundeshöchstleistungsrechner HLRB II am

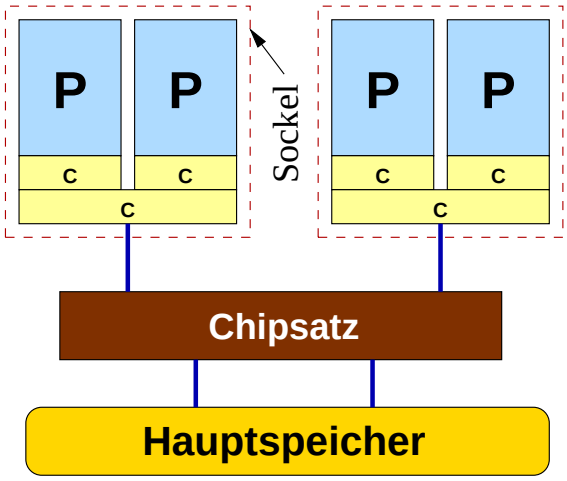

Abbildung 1.4: Beispiel für ein UMA-System, bei dem jeder Sockel eine separate Verbindung zum Chipsatz hat. Jeder Sockel beherbergt zwei Prozessorkerne (P), die über getrennte L1-Caches und einen gemeinsamen L2-Cache verfügen (C). (nach [6])

Leibniz-Rechenzentrum in München genannt [9]. Als Netzwerk zwischen den Lokalitätsdomänen kommt bei SGI-Systemen das routerbasierte *NUMALink* zum Einsatz (siehe Abbildung 1.5), welches leistungsfähig genug ist, um mehrere hundert Prozessoren zu bedienen. Am anderen Ende der Kosten- und Leistungsskala der NUMA-Rechner befinden sich Rechenknoten mit AMD Opteron Mehrkernprozessoren (siehe Abbildung 1.6). Die Lokalitätsdomänen, im Beispiel jeweils ein Prozessorsockel mit zwei Prozessoren (P), getrennten Caches (C) und gemeinsamer Speicherschnittstelle (SI), sind hier mittels *Hypertransport* (HT) verbunden.

Grundsätzlich verfügt in einem NUMA-System jede LD über eine Kommunikationsschnittstelle, die den Zugriff sowohl auf den direkt angebundenen als auch den über Netzwerk erreichbaren Speicher anderer Prozessoren transparent verwaltet. Eine weitere Aufgabe dieser Einheit ist es, Hauptspeicherinhalte und Cachezeilen der einzelnen Prozessoren kohärent zu halten. Ändert ein Prozessor eine Cachezeile, muss sichergestellt sein, dass das Ergebnis beim Zugriff auf diese Cachezeile durch einen anderen Prozessor (egal, ob sie bereits in seinem Cache liegt oder ob sie erst aus dem Hauptspeicher geladen werden muss) stets gültig ist. Selbstverständlich muss Cachekohärenz auch bei UMA gewährleistet sein, NUMA-Systeme verfügen jedoch nicht über eine zentrale Speicherverwaltung wie UMA-Systeme. Dies macht Cache-Kohärenz bei der Systementwicklung zu einer zentralen Problemstellung und schlägt sich in der Bezeichnung ccNUMA (cache-coherent NUMA) nieder.

Der große Vorteil der NUMA-Architektur ist die Skalierbarkeit, da eine Vergrößerung der Anzahl der Lokalitätsdomänen automatisch zu einer proportionalen Verbesserung der aggregierten Speicherbandbreite führt. Während es UMA-Rechner mit skalierbarer Speicherbandbreite derzeit nur bis maximal acht CPUs gibt, werden NUMA-Maschinen mit bis zu 1024 Prozessoren angeboten.

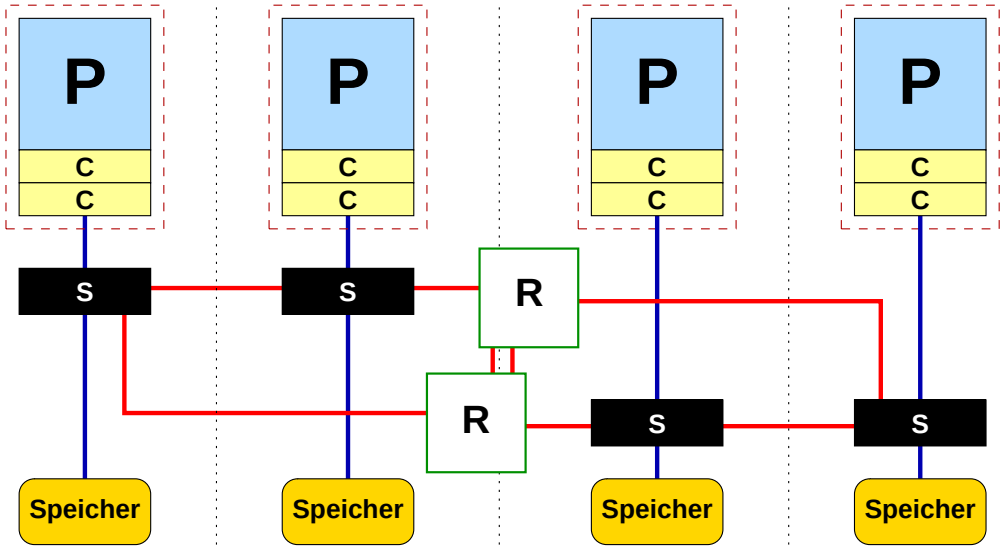

Abbildung 1.5: ccNUMA-System mit vier Lokalitätsdomänen, basierend auf einem gerouteten NUMALink Netzwerk. Über eine Kommunikationsschnittstelle (S) ist jeder Prozessorsockel an den Speicher und an einen Router (R) angekoppelt. (nach [6])

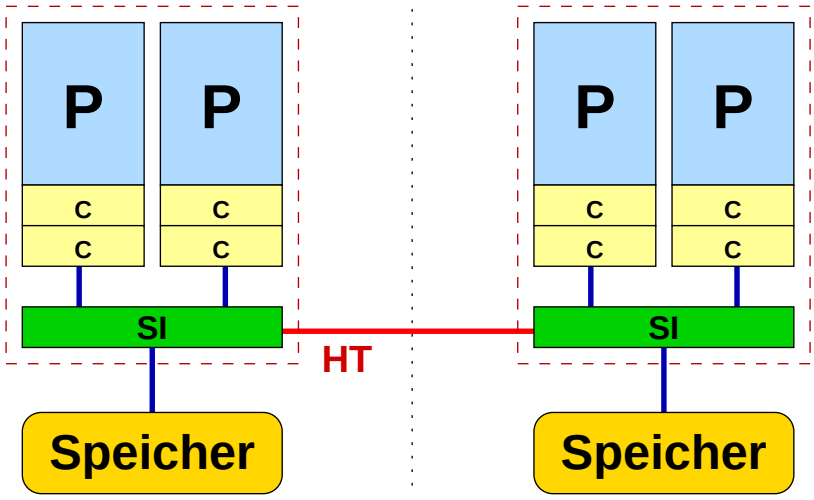

Abbildung 1.6: ccNUMA-System mit zwei Lokalitätsdomänen, die über ein Hypertransport (HT) Hochgeschwindigkeitsnetzwerk verbunden sind. (nach [6])

1.6 Beschreibung der Benchmarks

Im folgenden Abschnitt werden die verwendeten Benchmarks in der Grundversion vorgestellt. Themenspezifische Änderungen werden in den entsprechenden Kapiteln behandelt.

1.6.1 Vektortriade

Die Performance vieler Anwendungen wird nicht durch die Prozessorgeschwindigkeit, sondern durch die Speicherbandbreite und -latenz bestimmt [10]. Als Standardbenchmark gilt in diesem Zusammenhang die Vektortriade, die hohe Last auf dem Speichersubsystem erzeugt. Der Rechenkern besteht aus einer Schleife, in der Vektoren elementweise addiert und multipliziert werden (siehe Listing 1.1). Pro Iteration werden zwei Fließkommaoperationen ausgeführt, drei doppelt genaue Fließkomma-Worte geladen und ein Wort in den Speicher geschrieben. Um den Prozessor kontinuierlich mit Daten versorgen zu können, fällt also im Mittel ein Kommunikationsaufwand von zwei Worten pro Fließkommaoperation zwischen L1-Cache und Prozessorregistern an. Bei einem Woodcrest-Prozessor mit 3,0 GHz entspräche dies einem Datenvolumen von 192 GByte in der Sekunde. Die maximale Bandbreite liegt für diesen Prozessor zwischen 96 GByte/s (L1-Cache) und 10,6 GByte/s (Hauptspeicher). In keinem Fall kann also für die Vektortriade die maximale Performance (Peak) von 12 GFlop/s erreicht werden, da diese durch die Speicherbandbreite limitiert ist. Es können damit jedoch Richtwerte für die theoretisch mögliche Performance der einzelnen Speicherhierarchieebenen des Systems ermittelt werden. Befinden sich alle Daten im L1-Cache, d.h. jeder Vektor enthält bis zu 1024 Elemente, sind theoretisch 50 % der Peakperformance, also 6 GFlop/s, möglich. Im Speicher liegt die entsprechende Obergrenze bei etwa 660 MFlop/s. Damit ist es möglich, z.B. die Leistungsfähigkeit und das Optimierungspotenzial des Speicherzugriffsmusters und der Speicherbelegungsmethode abzuschätzen. Durch die Speicherbindung eignet sich die Vektortriade besonders zur Manifestierung der in dieser Arbeit behandelten Probleme mit NUMA-Systemen. Die naive Implementierung (siehe Abbildung 1.7) zeigt einen für cachebasierte Systeme typischen Performanceverlauf. Vor allem die Cacheperformance kann noch deutlich verbessert werden, aber von den theoretisch möglichen 6 GFlop/s werden in der Praxis trotzdem nur bis zu 4 GFlop/s gemessen. Ob dieser Wert wirklich erreicht werden kann, hängt von vielen Faktoren wie dem verwendeten Compiler und der gegenseitigen Lage der Arrays im Speicher ab. Die Untersuchung dieser Effekte ist jedoch nicht Teil der vorliegenden Arbeit.

Die Vektortriade besteht aus zwei geschachtelten Schleifen (siehe Listing 1.1). Die innere iteriert über die `N` Elemente jedes Vektors, mit denen arithmetische Operationen durchgeführt werden. Mit der äußeren Schleife kann die Rechendauer gesteuert werden. Die Rechendauer wird als sogenannte Wallclocktime gemessen, das ist die Zeit, die zwischen Beginn und Ende des Programmlaufes auf „der Uhr an der Wand“ vergeht.

Ungenauigkeiten bei der Zeitmessung wirken sich bei kurzen Laufzeiten stärker aus als bei langen. Kurze Laufzeiten treten bei kleinen Vektorlängen auf, also sollte `NITER` gerade bei kleinen `N` groß gewählt werden. Es empfiehlt sich, `NITER` dynamisch zu ermitteln, um für alle `N` eine Mindestlaufzeit sicherzustellen, die groß ist im Vergleich zu Messfehlern. Außerdem

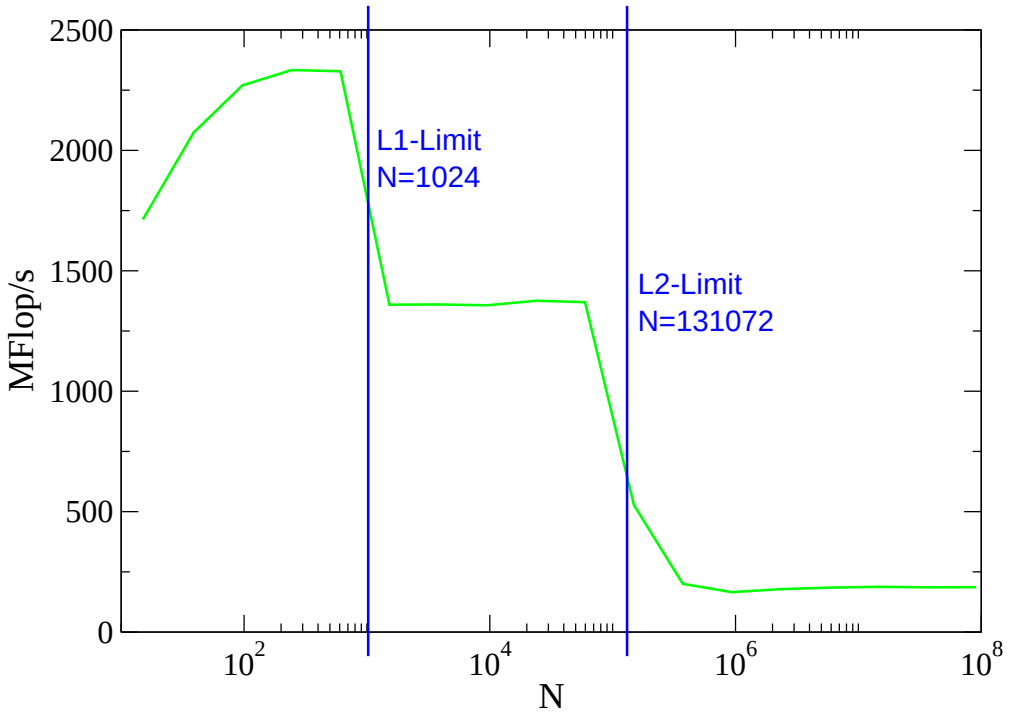

Abbildung 1.7: Typischer Performanceverlauf der Vektortriade (Woodcrest, seriell) über der Vektorlänge. Die Ausführungsgeschwindigkeit hängt stark von der Speicherhierarchieebene ab, aus der die Daten geladen werden müssen.

kann das Potenzial der Caches nur gemessen werden, wenn bereits geladene Daten erneut verarbeitet werden. Auch hierfür sorgt eine äußere Schleife mit großer Iterationszahl. Da die innere Schleife nicht vom Schleifenzähler der äußeren Schleife abhängt, wird ein hinreichend intelligenter Compiler letztere wegoptimieren. Der Aufruf der in eine separate Kompilationseinheit ausgelagerten Funktion `dummy()` mit allen vier Feldern als Parameter verhindert diese ungewollte Optimierung. Um die Messung nicht durch den Funktionsaufruf zu verfälschen, wird dieser durch eine unerfüllbare Bedingung maskiert. Diese darf jedoch wiederum nicht vom Compiler als unerfüllbar identifizierbar sein. Die Sprungvorhersagelogik des Prozessors sorgt dafür, dass nach wenigen Durchläufen der äußeren Schleife durch die „obscure“-Abfrage keine nennenswerten Kosten mehr entstehen [11]. Neben oben genannten Ungenauigkeiten bei der Zeitmessung müssen Schwankungen der Systemlast berücksichtigt werden. Ausgelöst werden diese zum Beispiel durch einen administrativen Cronjob mit großem Ressourcenbedarf, und können auch bei exklusiver Nutzung von Rechenknoten nicht vermieden werden. Um trotzdem verlässliche und repräsentative Benchmarkergebnisse zu erhalten, sollte die Gesamtmessung mehrfach wiederholt und der beste Wert weiterverwendet werden. Aus der ermittelten Laufzeit t_{wall} kann die Performance folgendermaßen berechnet werden:

$$P_{\text{triad}} = \frac{2 \cdot N \cdot N_{\text{iter}}[\text{Flop}]}{t_{\text{wall}}[s]} \tag{1.1}$$

Die übliche Einheit ist Fließkommaoperationen pro Sekunde (Flop/s).

Die Parallelisierung der Vektortriade mit OpenMP erfolgt durch Einfügen der Zeile

Listing 1.1: Vektortriade

```
for(int j = 1; j < NITER; ++j){
      for(int i = 0; i < N; ++i){
            a[i]=b[i]+c[i]*d[i];
      }
      if(obscure) dummy(a,b,c,d);
}
```

```
#pragma omp parallel for schedule(static)
```

vor der inneren Schleife. Damit werden die Schleifeniterationen auf die Threads verteilt und parallel abgearbeitet. Mit der Option `schedule` kann das Verteilungsschema bestimmt werden. Im vorliegenden Fall einer statischen Schedule ohne Angabe einer Blockgröße wird von jedem Thread ein etwa gleich großer (`chunk_size` = `N`/Threads), zusammenhängender Bereich abgearbeitet. Ein Vektorelement wird nur in genau einem Schleifendurchlauf gelesen bzw. geschrieben. Es bestehen also keine Datenabhängigkeiten zwischen den Threads, Kommunikation ist nicht erforderlich. Damit ist die Parallelisierung der Vektortriade trivial.

Read for Ownership (RFO) Bei den Bandbreitenkalkulationen zu Beginn des Kapitels wurde eine Besonderheit von Rechnersystemen mit hierarchischem Speicher nicht berücksichtigt. Damit ein Datum in ein Prozessorregister gelesen werden kann, muss es in der innersten Cacheebene vorhanden sein (siehe S. 3). Analog verhält es sich beim Schreiben, Registerinhalte werden stets in den Cache geschrieben. Zwischen Cache und Arbeitsspeicher können keine einzelnen Bytes, sondern nur komplette Cachezeilen übertragen werden. Deshalb muss der Prozessor eine Cachezeile zunächst laden, um die Berechtigung für das Ändern eines enthaltenen Datums zu erlangen (*read for ownership*, RFO). Daraus ergibt sich z.B. für die Vektortriade eine Verschlechterung der *Applikations-* oder *Codebalance*. Unter Applikationsbalance versteht man das Verhältnis von Speicherzugriffen zu arithmetischen Operationen. Statt der ursprünglich angenommenen vier Speicherzugriffe pro Iteration mit zwei Fließkommaoperationen, müssen nun fünf (3x Lesen, 1x RFO, 1x Schreiben) veranschlagt werden. Für speichergebundene Applikationen hat dies eine Verringerung der Performance zur Folge, bei der Vektortriade sind es theoretisch 25 %. Um RFO explizit verhindern zu können, bieten einige Architekturen spezielle Speicherbefehle. Diese *Nontemporal Stores* (auch *Streaming Stores* genannt) schreiben unter Umgehung der Cachehierarchie direkt in den Hauptspeicher. In einfachen Fällen kann der Compiler den Einsatz von Nontemporal Stores selbständig veranlassen, im Allgemeinen muss er aber darauf hingewiesen werden. Beim Intel-Compiler geht dies durch Einfügen der Compilerdirektive

```
#pragma vector nontemporal
```

vor der betreffenden Schleife. Gerade für Applikationen, die Datenströme verarbeiten, entsteht im speichergebundenen Fall durch das Umgehen des Caches beim Schreiben kein Nachteil, da das Lokalitätsprinzip hier nicht genutzt werden kann. Der Einsatz von Nontemporal

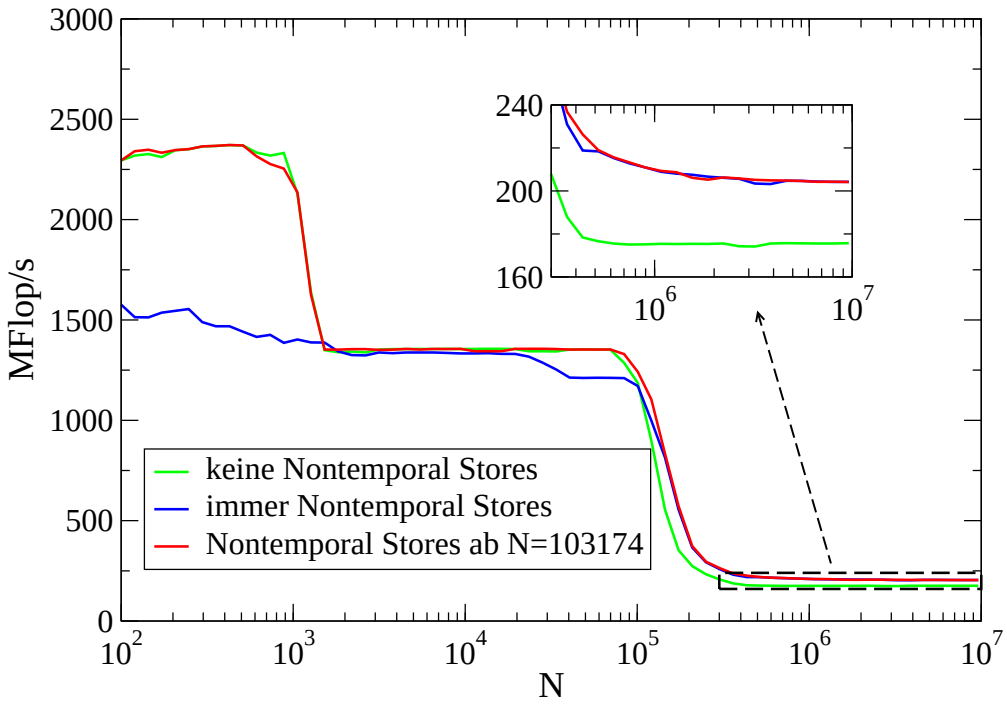

Abbildung 1.8: Performance der Vektortriade (Woodcrest) mit/ohne Nontemporal Stores, bzw. ab ca. 10^5 Vektorelemente.

Stores bei cachegebundenen Anwendungen ist allerdings kontraproduktiv, da damit Speicherbindung erzeugt wird. Abbildung 1.8 verdeutlicht potenzielle Vor- und Nachteile der Verwendung von Nontemporal Stores. Im Speicher steigt die Performance um 20 % von 175 auf 210 MFlop/s, im L1-Cache wird sie dagegen nahezu halbiert. Nontemporal Stores sollten deshalb abhängig von der Arraygröße aktiviert werden.

1.6.2 Relaxation

Der synthetische Vektortriaden-Benchmark ist hervorragend geeignet zur Visualisierung der Speicherhierarchie von Rechnersystemen und zur Bewertung von Speicherallokationsmethoden. Der Relaxationsbenchmark ist hinsichtlich seines Rechen- und Datenzugriffsmusters anwendungsnäher. Die zugrundeliegende Datenstruktur ist ein zweidimensionales Gitter, bei dem jedem Gitterpunkt ein Zahlenwert zugeordnet ist. Ausgehend von der Initialbelegung wird in jeder Iteration der Wert jedes Gitterpunktes als Mittelwert aus seinen vier Nachbarn neu berechnet. Eine Iteration, also die Aktualisierung aller Gitterpunkte, wird auch *Sweep* genannt. Als Folge der kontinuierlichen Mittelwertbildung gleichen sich die Werte der benachbarten Punkte immer mehr an. Für den Werteverlauf auf dem Gitter hat dies glättende, „entspannende“ Wirkung.

Ein Anwendungsgebiet für Relaxationsverfahren ist die numerische Lösung partieller Differentialgleichungen wie z.B. der *Laplace-Gleichung*,

$$\Delta T(x, y) = 0 \tag{1.2}$$

deren exakte Lösung häufig unpraktikabel oder unmöglich ist. Als iterative Löser kommen Relaxationsverfahren wie Gauß-Seidel zum Einsatz. Ein einfacher Anwendungsfall ist die Berechnung der Temperaturverteilung auf einer Platte, von der die Randwerte festgelegt sind. Jeder Gitterpunkt repräsentiert einen diskreten Temperaturwert auf der Platte. Die inneren Punkte (ohne Rand) können beliebig initialisiert werden. Abhänging von der gewünschten Genauigkeit, der Gitterauflösung und der Wahl der Startwerte, tritt nach einer gewissen Anzahl von Sweeps ein Zustand ein, bei dem sich die Gitterwerte von zwei aufeinanderfolgenden Durchläufen nicht mehr wesentlich verändern. Dieser Gleichgewichtszustand repräsentiert die theoretische Temperaturverteilung als Lösung der Laplace-Gleichung.

In der Praxis kommen Glättungsverfahren z.B. auch in der digitalen Bildverarbeitung und bei Mehrgittermethoden zum Einsatz. Letztere spielen in der numerischen Simulation, einem Bereich des High Performance Computing, eine zentrale Rolle. Die dort eingesetzten Varianten z.B. des Gauß-Seidel-Verfahrens basieren auf oben beschriebenem Prinzip. Die verwendeten Matrix-Datenstrukturen wie z.B. mehrdimensionale Felder sind allgegenwärtig im wissenschaftlichen Rechnen. Die Messwerte des Relaxationsbenchmarks erlauben also fundierte Rückschlüsse auf die Leistungsfähigkeit von Datenstrukturen und Rechnersystemen im Produktionsbetrieb. Die Codebalance von 1,5 Worten pro Flop (5x laden, 1x speichern, 3x addieren, 1x multiplizieren) mit, bzw. 1,25 Worten pro Flop ohne RFO hat eine Begrenzung der Performance durch die Bandbreite der verwendeten Speicherhierarchieebene zur Folge. Wie beim Vektortriaden-Benchmark kann damit die Tauglichkeit von Speicherstrukturen evaluiert werden.

Der Relaxationsbenchmark basiert auf einem Quell- und einem Zielfeld gleicher Dimension. Das Quellfeld muss mit den gegebenen Randwerten und beliebigen Werten im Inneren initialisiert werden. Der Wert eines Punktes im Zielfeld ergibt sich aus dem Mittelwert der Nachbarn des entsprechenden Punktes im Quellfeld. Durch Zeigertausch wechseln die beiden Felder nach jedem Sweep ihre Rolle, so dass die gerade neu ermittelten Werte stets weiterverarbeitet werden. Ein Sweep wird mit zwei geschachtelten Schleifen realisiert (siehe Listing 1.2). Die äußere iteriert über die Matrixzeilen, die innere über die Spalten der aktuellen Zeile. In beiden Fällen müssen die Schleifenzähler so angepasst sein, dass die Randwerte nicht verändert werden. Zeilenweiser Durchlauf der Matrix ist für diesen Benchmark nicht optimal. Durch Anpassung des Zugriffsmusters auf möglichst lokale Speicherzugriffe (z.B. durch sog. *Unrolling* von Schleifen) und damit gezielte Nutzung der schnellen Caches ist das Relaxationsverfahren seriell hochoptimierbar. Diese Feinabstimmungen sind jedoch nicht Teil dieser Arbeit, außerdem werden durch diese Veränderungen die Parallelisierung des Codes und die Interpretation von Messergebnissen erschwert. Die Anzahl der Sweeps wird in der Praxis von einigen Faktoren bestimmt (s.o.). Für Benchmarkzwecke wurde statt eines Abbruchkriteriums eine feste Anzahl auszuführender Sweeps definiert. Die Parallelisierung des Codes erfolgt durch Einfügen des OpenMP-Pragmas vor die Schleife, die die Matrixzeilen abarbeitet. Gegenüber einer vertikalen Aufteilung des Gebietes hat dies den Vorteil, dass die Zeilen nicht getrennt werden. Eine Zeile ist ein physikalisch zusammenhängender Speicherblock. Hier kann der Compiler besondere Optimierungen wie z.B. Vektorisierung anwenden [26]. Für hinreichend große Systeme trifft dies auch für eine vertikale Aufteilung zu. In Kapitel 4 werden beide Varianten untersucht.

Listing 1.2: Relaxation

```
for(int k = 0; k < SWEEPS; ++k){
#pragma omp parallel for schedule(static)
      for(int j = 1; j < ROWS-1; ++j){
            for(int i = 1; i < COLS-1; ++i){
                  matrix1[i][j] = ( matrix2[i-1][j] + matrix2[i][j+1]
                              + matrix2[i][j-1] + matrix2[i+1][j] ) * 0.25;
            }
      }
      tmp = matrix1; matrix1 = matrix2; matrix2 = tmp;
}
```

1.6.3 Benchmarkumgebung

Rechnersysteme

In Tabelle 1.1 werden die beiden für Benchmarks verwendeten Rechnersysteme kurz beschrieben. Zum Einsatz kamen ein UMA-System mit Intel Woodcrest Prozessoren und ein ccNUMA-System mit Opteron-Prozessoren von AMD. Bei Ersterem handelt es sich um einen Knoten des Ende 2006 am RRZE in Betrieb genommenen Clusters, das in den letzten beiden Ausgaben der Liste der weltweit 500 schnellsten Supercomputer [12] auf Rang 124 respektive 273 platziert war. Die ccNUMA-Maschine ist ein zu Benchmarkzwecken beschafftes Stand-Alone-System. Im weiteren Text werden die beiden Rechner anhand der Namen ihrer Prozessoren referenziert.

Compiler

Die Benchmark-Codes wurden mit dem Intel C/C++-Compiler für 64-Bit Anwendungen in der zu Beginn der Arbeit aktuellen Version 9.1.049 übersetzt. Mit der Compileroption „-O3“ wurde die höchste Optimierungsstufe aktiviert. Um die Vektorisierung von Schleifen durch den Compiler und die damit zusammenhängende Nutzung des SIMD-Befehlssatzes der Prozessoren zu ermöglichen, wurde auf dem Woodcrest-System die Compileroption „-xP“ (für SSE3) und auf dem Opteron-System die Option „-xW“ (für SSE2) spezifiziert.

	Woodcrest	**Opteron**
Prozessoren	Intel Xeon 5160 „Woodcrest“ Dualcore, 3,0 GHz	AMD Opteron 8220 SE Dualcore, 2,8 GHz
Peak Performance pro Kern	12 GFlop/s	5,6 GFlop/s
Arbeitsspeicher	8 GB	16 GB
Cache	32 kB L1 pro Kern 4 MB L2 pro Chip	64 kB L1 und 1 MB L2 pro Kern
Sockel	2	4 (Sockel F)
Lokalitätsdomänen	1	4
Theoretische Speicherbandbreite pro Sockel	10,6 GB/s	10,6 GB/s
Theoretische Speicherbandbreite des Systems	21,2 GB/s	42,4 GB/s
Stream-Bandbreite	6,5 GB/s	18,3 GB/s

Tabelle 1.1: Hardwarecharakteristiken der Benchmarksysteme

2 NUMA-Problematik

Der physikalisch verteilte Hauptspeicher in ccNUMA-Rechnern erfordert besondere Aufmerksamkeit bei der Entwicklung von HPC-Applikationen. Die Abbildung vom heterogenen physikalischen auf den homogenen virtuellen Speicher ist für Programmierer und Compiler transparent. Um teuere Zugriffe auf nichtlokalen Speicher vermeiden zu können, ist deshalb neben der genauen Kenntnis des Datenzugriffsmusters ein grundlegendes Verständnis der Speicherverwaltung erforderlich. Dieses soll im folgenden Abschnitt vermittelt werden. Vorher werden die auftretenden Bottlenecks lokalisiert.

Der Umgang mit ccNUMA-Systemen birgt zwei Arten von Schwierigkeiten. Die bereits angesprochen Zugriffe eines Prozessors auf Speicher in einer anderen Lokalitätsdomäne (siehe Abbildung 2.1) führen trotz schneller Netzwerke zu signifikanten Performanceeinbußen gegenüber rein lokalen Zugriffen. Abbildung 2.2 zeigt dieses *Lokalitätsproblem* anhand des Triaden-Benchmarks.

Bei parallelen Codes kommt zusätzlich ein *Bandbreitenproblem* hinzu (Abbildung 2.3). Greifen zwei Prozessoren, die in verschiedenen Lokalitätsdomänen liegen, gleichzeitig auf Speicher in einer Lokalitätsdomäne zu, so begrenzt die Bandbreite des Speicherkanals die Performance (siehe Abbildung 2.4). Gewöhnlich ist eine Lokalitätsdomäne den Bandbreitenanforderungen zweier oder mehr Prozessoren nicht gewachsen. Abhilfe schafft in beiden Fällen die Beschränkung jedes Prozessors auf lokale Speicherzugriffe, falls dies der verwendete Algorithmus erlaubt.

2.1 Lösungsansatz Page Placement

Um nichtlokale Speicherzugriffe möglichst gering zu halten, muss bereits beim Anlegen von Datenstrukturen das spätere Zugriffsmuster berücksichtigt werden. Daten müssen in der Lokalitätsdomäne des Prozessors platziert werden, der sie verarbeitet. Hierfür wird eine Eigenschaft der virtuellen Speicherverwaltung genutzt: Zur Assoziation von virtuellen zu physikalischen Speicherseiten führt das Betriebssystem für jeden Prozess eine Seitenzuordnungstabelle. Schreibt ein Thread ein Element einer bisher ungenutzten Seite des virtuellen Speicherbereichs, so wird dieser mit einem neuen Eintrag in der Seitentabelle eine physikalische Speicherseite im lokalen Speicher des Prozessors, auf dem der Thread läuft, zugeordnet. Der Prozessor, der eine Speicherseite zuerst berührt, hat diese dann in seinem lokalen Speicher. Dieser Vorgang wird *First-Touch-Mapping* genannt und bildet die Basis für NUMA-gerechte Speichernutzung. Konkret müssen die Daten also nach dem gleichen Muster initialisiert werden wie sie später verarbeitet werden sollen. Bei Standarddatentypen ist Allokation, z.B. mittels `new()` oder `malloc()` nicht ausreichend, da hier nicht schreibend

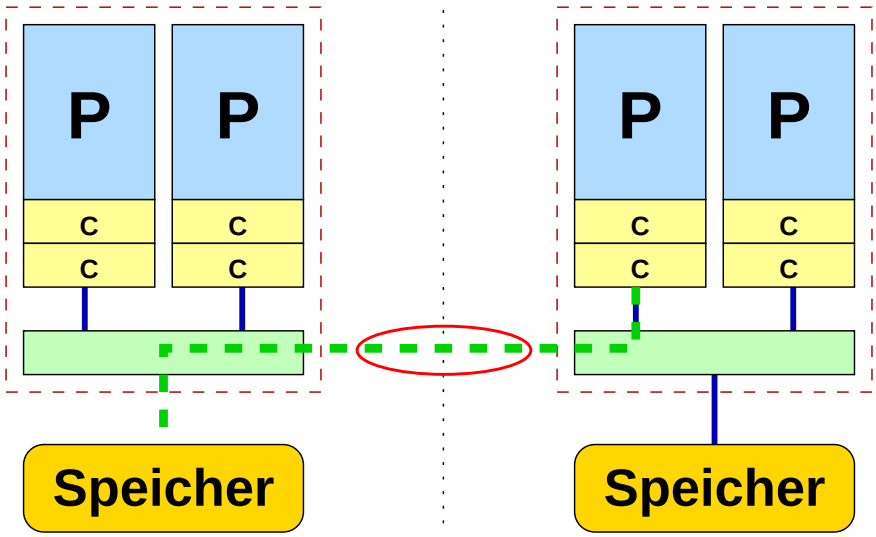

Abbildung 2.1: Lokalitätsproblem (nach [6])

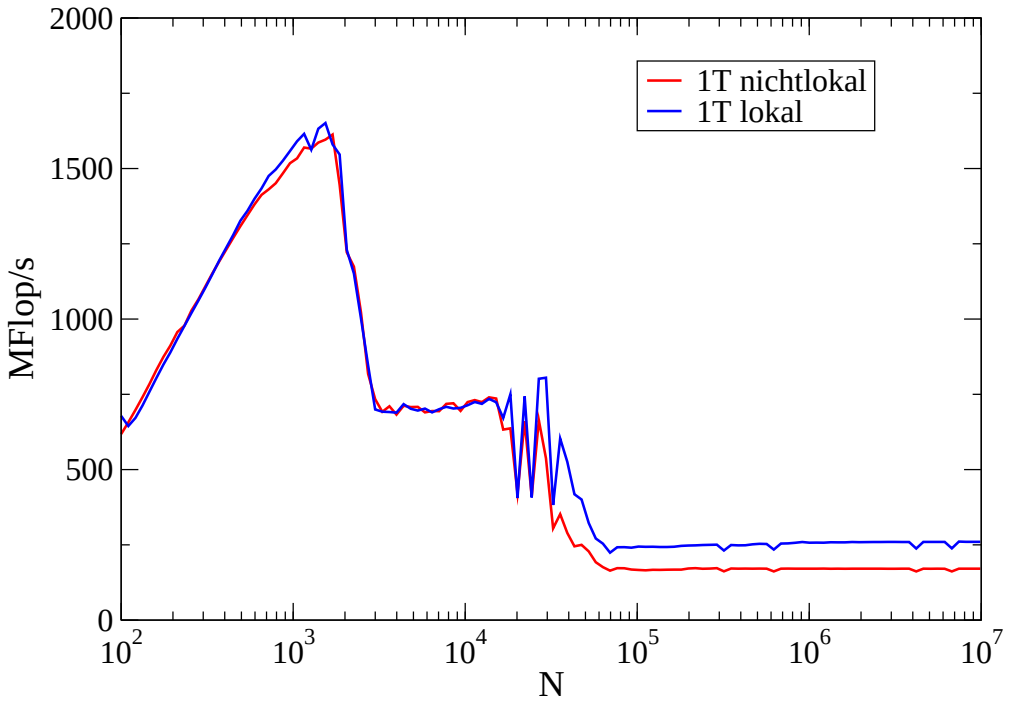

Abbildung 2.2: Bei ausschließlich nichtlokalen Speicherzugriffen ist die Performance der Vektortriade im Speicher um 30 % geringer als bei ausschließlich lokalen. Der lokale Arbeitsspeicher wurde durch einen großen Datenblock belegt, so dass die Vektoren in nichtlokalem Speicher platziert wurden. (Messungen auf dem Opteron-System)

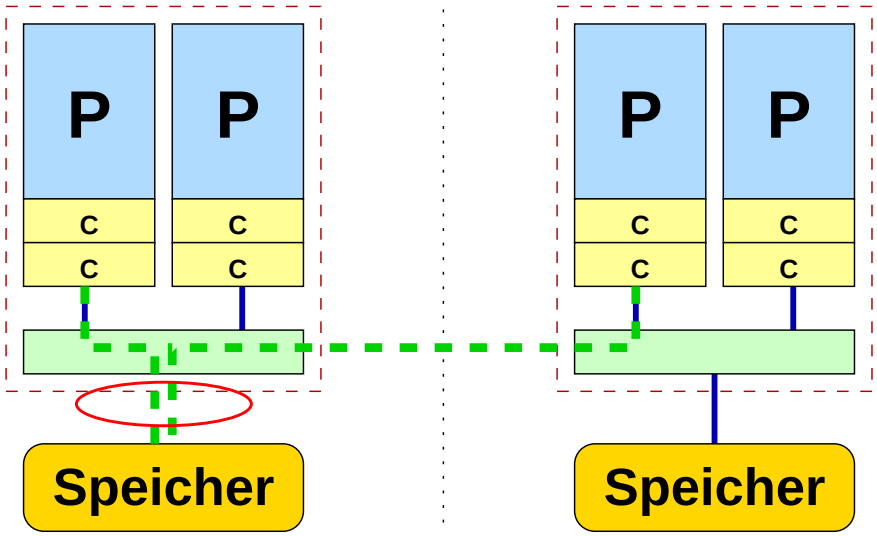

Abbildung 2.3: Bandbreitenproblem (nach [6])

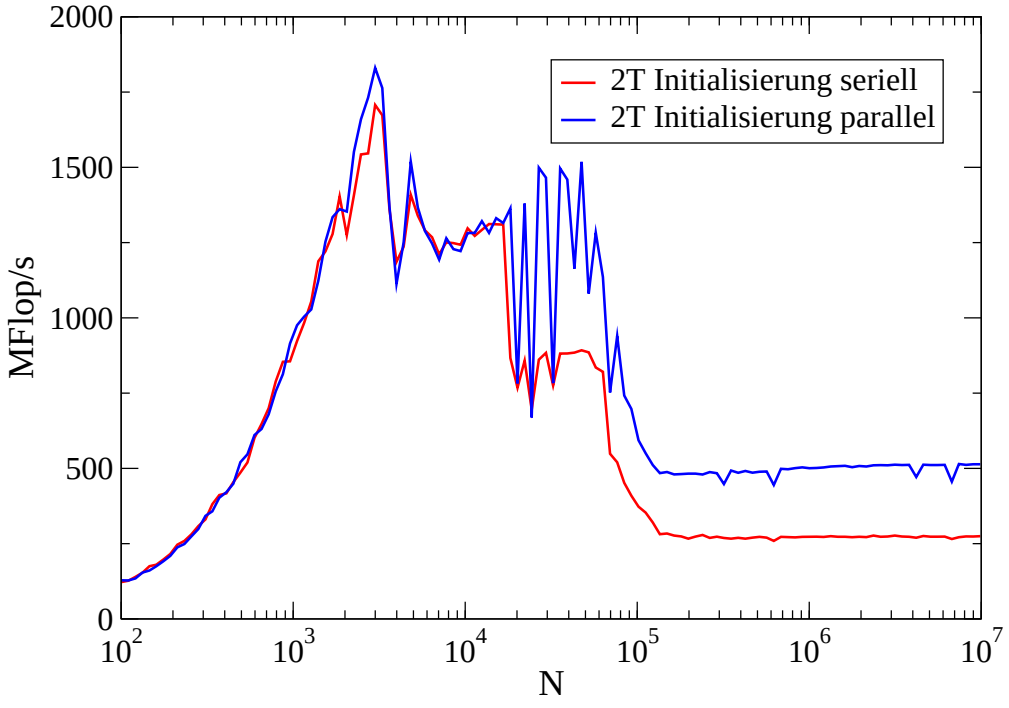

Abbildung 2.4: Durch parallele Initialisierung kann die Performance der Vektortriade im Speicher fast verdoppelt werden. (Messungen auf dem Opteron-System)

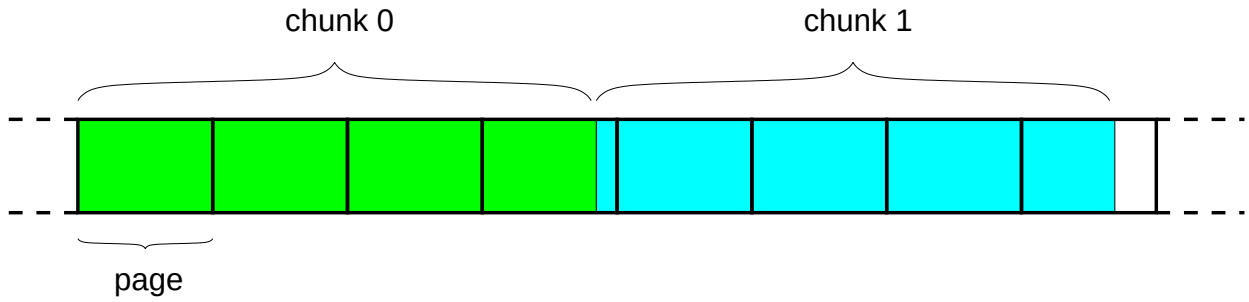

Abbildung 2.5: Zuordnung von Speicherseiten (page) zu OpenMP-Arbeitspaketen (chunk)

auf den Speicher zugegriffen wird. Es kann gezeigt werden, dass auch in Fällen, in denen das Datenzugriffsmuster erratisch oder nicht vorhersehbar ist, eine gleichmäßige Verteilung der Daten auf die Lokalitätsdomänen der beteiligten Threads bessere Performancewerte liefert, als rein serielle Initialisierung [13].

Bei der Entwicklung NUMA-optimierter Software muss berücksichtigt werden, dass das Betriebssystem nur dann Speicherseiten in der richtigen Lokalitätsdomäne zuordnen kann, wenn dort noch genug Platz vorhanden ist. Dieser Platz kann z.B. durch vom Betriebssystem angelegte Pufferbereiche für Datei-Ein-/Ausgabe belegt sein, was trotz korrekter Initialisierung zu falschem Placement und somit nichtlokalen Zugriffen führt.

In Listing 2.1 ist der Code der Vektortriade mit paralleler Initialisierung abgebildet, der zu den Messergebnissen in Abbildung 2.4 führt. Wie bei der Berechnung kommt bei der Initialisierung eine einfache Schleife über alle Vektorelemente zum Einsatz, die mit statischer Schedule parallelisiert wird. Jeder Thread bekommt also, abgesehen von Randeffekten (siehe Abbbildung 2.5), die Vektorelemente in seinen lokalen Speicher, die er später verarbeitet. Allerdings sollte das Datenvolumen für jeden Thread mindestens einige Speicherseiten[1] umfassen, da sonst durch die negative Auswirkung falsch platzierter Randwerte die Vorteile der Parallelisierung relativiert werden. Dies wird besonders deutlich, wenn man den Extremfall betrachtet, dass das Gesamtdatenvolumen der Messung nur eine Speicherseite groß ist. Dann befinden sich alle Daten im lokalen Speicher des Threads, der die Seite zuerst berührt. Alle Speicherzugriffe von Threads aus anderen Lokalitätsdomänen sind damit nichtlokal. Je mehr Speicher jeder Thread anlegt, desto geringer ist die negative Auswirkung von Seiten, die von mehreren Threads genutzt werden. Obwohl eine einzelne Speicherseite natürlich im Cache eines jeden modernen Mikroprozessors Platz findet, gibt es dennoch Situationen, in denen die Datenverteilung über die einzelnen Threads in sehr kleinen, stark verschränkten Einheiten erfolgt. In diesem Fall ist eine Reorganisation der Datenhaltung angebracht, um NUMA-gerechtes Placement erreichen zu können.

Bei den vorangegangenen Betrachtungen wurde davon ausgegangen, dass die beiden parallelen Schleifen für Initialisierung und Berechnung nach dem gleichen Muster parallelisiert werden, d.h. bei beiden wurde die gleiche OpenMP-Schedule verwendet. Dies ist jedoch nicht ausreichend, da es bei den Schedules `dynamic` und `guided` Laufzeitabhängigkeiten ([14] S. 36) gibt, die dazu führen, dass im Allgemeinen die Aufteilung der Schleifeniterationen auf Threads in jeder parallelen Region unterschiedlich ist. Reproduzierbar ist der Ablauf nur bei

[1] Eine Speicherseite hat unter Linux üblicherweise die Größe von 4 kB.

Listing 2.1: Vektortriade mit NUMA-gerechter Initialisierung

```
double* a = new double[N];
double* b = new double[N];
double* c = new double[N];
double* d = new double[N];

#pragma omp parallel for schedule(static)
for(int i = 0; i < N; ++i){
      a[i]=b[i]=c[i]=d[i]=1.0;
}
for(int j = 1; j < NITER; ++j){
#pragma omp parallel for schedule(static)
      for(int i = 0; i < N; ++i){
            a[i]=b[i]+c[i]*d[i];
      }
      if(obscure) dummy(a,b,c,d);
}
```

statischer (`static`) Schedule, alle weiteren Betrachtungen setzen diese deshalb voraus. Der Iterationsraum wird hier möglichst gleichmäßig auf die Threads aufgeteilt (siehe Abbildung 2.5), und jeder Thread bekommt genau ein Paket zugewiesen. Die Größe der Arbeitspakete (`chunk_size`) kann auch explizit festgelegt werden. Darauf bezieht sich der letzte Abschnitt dieses Kapitels.

Parallele Initialisierung ist unbedingt erforderlich, um korrektes NUMA-Placement zu erreichen. Bei einfachen Datentypen, wie zum Beispiel den oben gezeigten `double`-Feldern, ist dies auch leicht umzusetzen. Sollen jedoch Felder von C++-Klassen oder STL-Container NUMA-gerecht angelegt werden, sind besondere Maßnahmen erforderlich, die im folgenden Kapitel genau beleuchtet werden. Schwierigkeiten gibt es ebenfalls bei der Nutzung globaler Datenfelder. Diese werden bereits vor dem Aufruf der `main()`-Funktion und damit auch vor allen parallelen Regionen in einer einzigen Lokalitätsdomäne erzeugt.

2.2 Process Pinning

Die bisherigen Überlegungen zum First-Touch-Placement basieren auf der Annahme, dass die Threads paralleler Programme auf die „richtigen“ Prozessoren des Systems verteilt werden. Bei Anwendungen, die eventuell vorhandene gemeinsame Caches nutzen, lägen diese Kerne auf einem Sockel. Speichergebundene Anwendungen würden dagegen davon profitieren, auf verschiedenen Sockeln zu laufen. Der Prozess-Scheduler des Betriebssystems hat jedoch keinen Einblick in die Ressourcenanforderungen der Threads und platziert diese deshalb im Allgemeinen nicht ideal. Die Positionierung hängt vielmehr von der Systemauslastung ab. Deshalb kann die initiale Thread-Prozessor-Zuordnung von Programmstart zum Programmstart variieren, was Performanceschwankungen zur Folge hat. Auch zur Laufzeit können

Listing 2.2: Pinning von Threads auf Prozessoren mit PLPA

```
#include <plpa.h>
#include <omp.h>

plpa_cpu_set_t mask;
int my_proc_id[] = {1,2,3,4,5,6,7,0}

#pragma omp parallel
{
      PLPA_CPU_ZERO(&mask);
      int thread = omp_get_thread_num();
      PLPA_CPU_SET(my_proc_id[thread], &mask);
      PLPA_NAME(sched_setaffinity)((pid_t)0, (size_t)32, &mask);
}
```

Threads auf andere Prozessoren verschoben werden, z.B. durch administrative Cronjobs, die hohe Systemlast erzeugen. Zusätzlich zur Beeinträchtigung des Placement geht durch einen Prozessorwechsel auch der Cacheinhalt verloren, was insbesondere bei cachebasierten Programmen u.U. zu einem messbaren Performanceeinbruch führt.

Diese Instabilität kann jedoch verhindert werden, indem beim Programmstart jeder Thread einem Prozessor fest zugeordnet wird (*Pinning*). Unter Linux kann dies mit den Kernelfunktionen `sched_getaffinity()` und `sched_setaffinity()` gesteuert werden, deren API jedoch von der Distribution, der Kernel- und der `glibc`-Version abhängt. Eine einheitliche und damit portable Lösung bietet die *PLPA*[2]-Bibliothek [15], die intern die jeweils vorhandene Version der genannten Funktionen nutzt.

Die Prozessorkerne werden vom Betriebssystem nummeriert. Pinning gestaltet sich also als die Zuordnung von Threadnummern zu Prozessor-IDs. In Codebeispiel 2.2 wird die Anwendung von PLPA demonstriert. Die Variable `mask` repräsentiert eine Bitmaske, die mit `PLPA_CPU_ZERO()` initialisiert werden kann. Das Bit für den Prozessor[3], auf dem der Thread laufen soll, muss mit eins belegt werden. Dafür gibt es die Funktion `PLPA_CPU_SET()`. Mit `PLPA_NAME()` erfolgt der Aufruf der `sched_setaffinity`-Funktion des Betriebssystems. Um die Bedienung zu erleichtern wurde das Feld `my_proc_id[]` eingeführt. Es muss mindestens so viele Einträge umfassen wie es Threads geben soll. Ein Thread wird auf den Prozessor gepinnt, dessen ID an „seiner“ Stelle im Array steht. Im Beispiel wird der Thread mit der Nummer 3 auf Prozessor 4 beschränkt.

Damit `my_proc_id[]` sinnvoll belegt werden kann, muss die Zugehörigkeit der Prozessoren zu Lokalitätsdomänen bekannt sein. Sollte diese nicht bekannt sein, kann sie unter Ausnutzung der beschriebenen Bandbreitenprobleme ermittelt werden. Bei speichergebundenen Applikationen ist die Aufteilung von Daten und Arbeit auf mehrere Lokalitätsdomänen und damit

[2] Portable Linux Processor Affinity

[3] Ein Thread kann auch auf einen aus mehreren Prozessoren bestehenden Satz beschränkt werden.

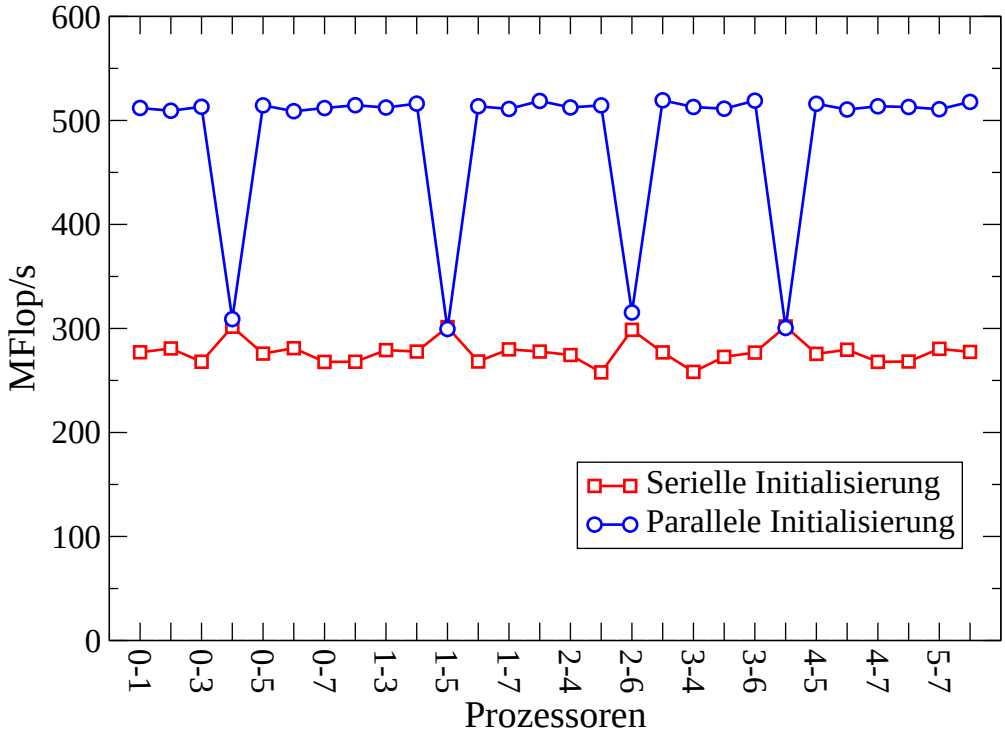

Abbildung 2.6: Performance der Vektortriade für alle Prozessornummerpermutationen. (Opteron, 2 Threads, $9 \cdot 10^6$ Elemente pro Vektor)

Prozessorsockel angebracht, da so die effektive Speicherbandbreite erhöht wird. Die Performanceeinbrüche der oberen Kurve in Abbildung 2.6 lassen also darauf schließen, dass hier beide Threads auf einen Sockel gepinnt wurden. Die Performance bricht in diesem Fall um fast 50 % auf die bei serieller Initialisierung der Arrays ein. Die Prozessoren 0 und 4, 1 und 5, 2 und 6, 3 und 7 befinden sich also jeweils auf einem Sockel. Die Zuordnung im Codebeispiel ist also für die Vektortriade geeignet. Bei allen im Rahmen dieser Arbeit durchgeführten Benchmarks wurden die Threads auf verschiedene Sockel verteilt, sofern die Anzahl der Threads nicht größer als die Anzahl vorhandener Sockel war.

Aus Abbildung 2.6 kann eine weitere interessante Tatsache abgeleitet werden. Die Performance bei serieller Initialisierung steigt um ca. 10 %, wenn beide Threads auf dem selben Sockel liegen. Ein einzelner Prozessor kann also den Speicherbus nicht voll auslasten.

2.3 Variation der OpenMP-Blockgröße

Damit Initialisierungs- und Arbeitsschleifen nach identischem Muster parallelisiert werden, ist eine OpenMP-Schedule vom Typ `static` erforderlich. Jeder Thread bearbeitet bearbeitet einen bestimmten Bereich im Iterationsraum, dessen Größe von der Schleifenlänge und der Anzahl der beteiligten Threads abhängt. Der OpenMP-Standard [14] sieht die Möglichkeit vor, die Größe dieser Arbeitspakete (Parameter `chunk_size`), also die Anzahl der aufeinanderfolgenden Schleifeniterationen für jeden Thread, zu variieren. Beginnend am Schleifen-

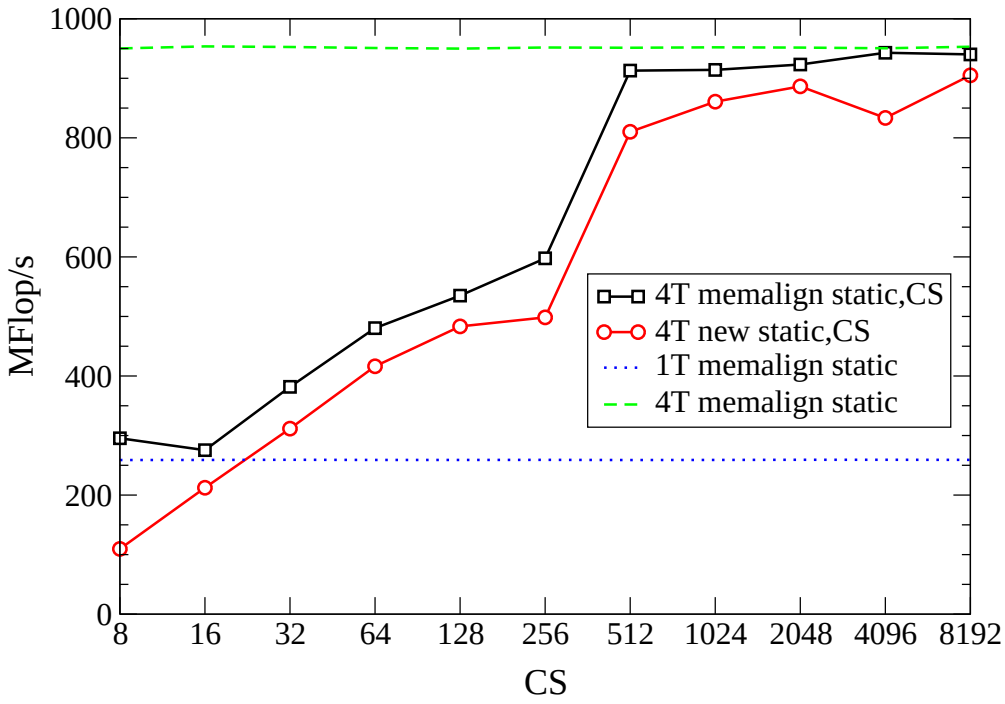

Abbildung 2.7: Performance der Vektortriade (Opteron, 4 Threads) bei einer Vektorlänge von 10^7 für verschiedene Blockgrößen (CS).

anfang bekommt jeder Thread abwechselnd einen Bereich dieser Größe zugewiesen, bis der Iterationsraum abgedeckt ist. Die Wahl einer kleinen `chunk_size` ermöglicht eine feingranulare Verteilung der Iterationen auf die Threads. Dies kann z.B. bei Schleifen mit ungleichem Arbeitsaufwand pro Iteration sinnvoll sein, da dadurch im Mittel eine gleichmäßige Lastverteilung gewährleistet ist.

In Abbildung 2.7 wird die Performance der Vektortriade für verschiedene Blockgrößen (`chunk_size`) verglichen. Die parallelisierten Schleifen iterieren bei der Vektortriade über Arrays. Es existiert hier ein direkter Zusammenhang zwischen der Blockgröße und der Größe des zusammenhängenden Speicherblocks, den ein Thread bearbeitet. Beispielsweise entspricht die `chunk_size` von 8 im Fall des `double`-Arrays 64 Bytes, der Länge einer Cachezeile. Ein kleinerer Wert ist also nicht sinnvoll. Auch bei Werten unter 512 sind deutliche Performanceeinbußen gegenüber der Variante mit reiner `static`-Schedule zu verzeichnen. Ab dieser Grenze füllt jedoch das einem Thread zugeordnete Datenvolumen mindestens eine Speicherseite (mit der üblichen Größe von 4kB), und kann, da NUMA-Placement auf Speicherseitenebene stattfindet, komplett im lokalen Speicher abgelegt werden. Bei kleineren Chunks teilen sich mindestens zwei Threads eine Speicherseite. Dies führt zwangsläufig zu nichtlokalen Zugriffen, was den Performanceverlauf erklärt.

Der virtuelle Speicher ist in Seiten segmentiert, d.h. die Anfangsadresse einer Speicherseite ist ein ganzzahlig Vielfaches der Seitengröße. Damit ein Datenblock mit der Größe einer Speicherseite in eine einzige Speicherseite passt, muss er an einer solchen Adresse ausgerichtet sein. Ein komfortables Werkzeug hierfür ist die Funktion `memalign()`, ein in `memalign.h`

definierter Wrapper um `malloc()`. Damit wird ein Array, wie es bei den Messungen für Abbildung 2.7 zum Einsatz kam, folgendermaßen allokiert:

```
int arraysize = 10000000;
double *a = (double *)memalign(4096, arraysize);
```

Mit `new[]` angelegter Speicher ist im Allgemeinen nicht an Speicherseiten ausgerichtet. Deshalb kommt es zu Überlappung von Speicherbereichen und -seiten verschiedener Threads und damit zu nichtlokalen Speicherzugriffen. Im Beispiel kostet dies etwa 10 % der Ausführungsgeschwindigkeit.

3 NUMA-Optimierung von C++-Code für High Performance Computing

Felder aus Standarddatentypen wie z.B. `int`, `double` oder `char` werden aufgrund ihrer großen Flexibilität auch in C++-Programmen häufig eingesetzt, obwohl es hier weitaus robustere Möglichkeiten gibt. In Abschnitt 2.1 wurde die Methode des First-Touch-Placement eingeführt, mit der Datenlokalität auf Systemen mit nichteinheitlichem Speicher hergestellt werden kann. Grundlegend hierfür ist, dass Speicherseiten im lokalen Speicher des Threads angelegt werden, der sie zuerst liest oder beschreibt (First-Touch-Mapping). Umgesetzt wird dies mit einer parallelen Schleife, die Daten nach dem gleichen Muster initialisiert, nach dem sie anschließend verarbeitet werden (siehe Listing 2.1). Die Initialisierung geschieht bei komplexeren C++-Datenstrukturen wie Objekten oder STL-Containern automatisch, korrektes Page Placement erfordert also tiefere Eingriffe in die Speicherverwaltung.

3.1 C++-Objekte

Bei der Instanziierung von C++-Objekten wird nach der Allokation des notwendigen Speichers direkt der Standardkonstruktor aufgerufen. Die Elemente eines Feldes aus C++-Objekten werden also im Allgemeinen beim Aufruf von `new()` bereits initialisiert, das heisst mit einem Defaultwert beschrieben. Dies geschieht seriell, ein Feld aus C++-Objekten liegt also im lokalen Speicher des Threads, der `new[]()` ausführt [16]. Ein möglicher Ausweg wäre ein Defaultkonstruktor, der die Klassenvariablen nicht initialisiert. Allerdings ist diese Einschränkung im Allgemeinen nicht erfüllbar, da Memberfunktionen sich häufig auf initialisierte Objekte verlassen.

Tatsächlich führt eine genauere Untersuchung der Funktionsweise des `new`-Ausdrucks zur Lösung des Problems. Bei einem Konstrukt wie

```
string * sp = new string("initialized");
```

finden drei Schritte statt [17]:

- Zunächst wird ein Speicherbereich allokiert, der groß genug ist, um das zu erzeugende Objekt zu fassen. Dazu wird die Allokationsfunktion `operator new` aufgerufen (siehe Abbildung 3.1).
- Danach wird mit dem Standardkonstruktor ein Objekt erzeugt.

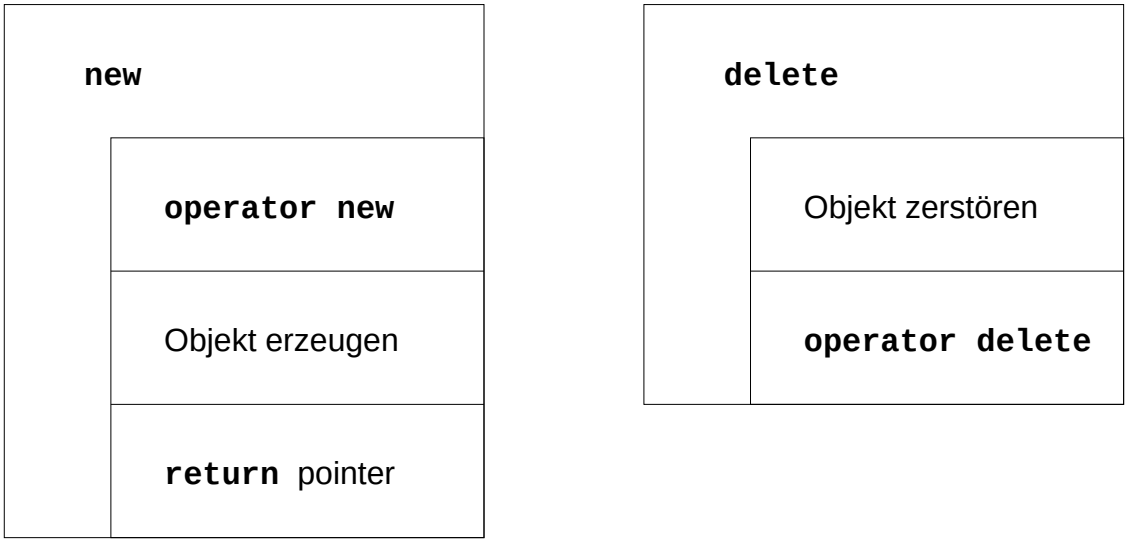

Abbildung 3.1: Bei `new` und `delete` ist die Namensgebung in C++ verwirrend. Man unterscheide zwischen `new`-Expression und `operator new`, bzw. zwischen `delete`-Expression und `operator delete` [17]. Einige Quellen nutzen für `new`-Expression bzw. `delete`-Expression die Bezeichnungen `new`-Operator bzw. `delete`-Operator. Aufgrund der größeren Verwechslungsgefahr mit `operator new / operator delete` wird diese Nomenklatur nicht verwendet.

- Schließlich wird ein Zeiger auf den Beginn des Objekts im Speicher an den Aufrufer zurückgegeben.

Komplementär dazu werden durch den Aufruf

```
delete sp;
```

die folgenden beiden Maßnahmen eingeleitet:

- Auf das Objekt an der Speicherstelle `sp` wird der Destruktor angewandt.
- Die Deallokationsfunktion `operator delete` gibt den Speicherbereich frei.

Zentraler Zweck von `operator new` und `operator delete` ist die Speicherallokation bzw. -deallokation im Rahmen von `new`-Expressions. Beide sind jedoch reguläre, in der C++-Standardbibliothek definierte Funktionen. Sie können also auch direkt aufgerufen werden um, ähnlich zu `malloc` und `free`, uninitialisierten, typlosen Speicher zu allokieren bzw. um Speicher freizugeben. Die bedeutendere Schlussfolgerung ist jedoch, dass sie, global oder innerhalb einer Klasse, *überladen* werden können. Damit kann das Verhalten von `new` und `delete` auch für bestehenden Code leicht angepasst werden. Wenn für eine Klasse ein überladener `operator new` oder `operator delete` vorhanden ist, wird dieser vom Compiler gegenüber der globalen Version vorgezogen. Innerhalb einer solchen Klasse können die globalen Operatoren mit der expliziten Angabe des Namensraumes `std` erreicht werden. Um mit einem überladenen `operator new` allokierten Speicher wieder konsistent freigeben zu

können, muss auch `operator delete` überladen werden. Beide müssen korrespondierende Allokations- bzw Deallokationsfunktionen wie z.B. `malloc()` und `free()` verwenden. Für `new[]` und `delete[]`, also für Felder aus C++-Objekten, gelten die bisherigen Überlegungen analog. Hierzu können die Funktionen `operator new[]` und `operator delete[]` überladen werden. Die Objekte werden sequenziell im Speicher konstruiert, beginnend am Anfang des dafür vorgesehenen Speicherbereichs. Die Destruktion erfolgt in umgekehrter Reihenfolge. Falls nicht gesondert beschrieben, gelten auch die weiteren Ausführungen für beide Arten.

Folgender Interfacedefinition müssen überladene Versionen von `operator new` und `operator delete` genügen, die innerhalb einer `new` bzw. `delete` Expression genutzt werden sollen. `operator new` hat als Rückgabewert einen typlosen Zeiger, also `void*`. Der Parameter vom Typ `size_t` wird von der `new`-Expression mit der Anzahl zu allokierender Bytes belegt. Dieser Wert muss nicht exakt mit der Objektgröße oder der Feldgröße übereinstimmen. Tests auf verschiedenen Systemen haben gezeigt, dass einige Compiler etwas mehr Speicher anfordern als erwartet, um dort z.B. Verwaltungsinformationen ablegen zu können. Aus diesem Grund unterscheidet sich möglicherweise auch die von `operator new` zurückgelieferte Adresse von der, an der sich das erste Objekt befindet, abhängig davon, ob die Zusatzinformationen am Beginn oder am Ende des Speicherbereichs platziert werden. Auch der erste Parameter von `operator delete`, ein Zeiger vom Typ `void`, der von der `delete`-Expression auf den Beginn des freizugebenden Speicherbereichs gesetzt wird, unterliegt dieser Unwägbarkeit. Dessen Inhalt muss demnach nicht zwingend identisch sein mit der Adresse des (ersten) Objekts im Feld. Ein optionaler zweiter Parameter von `operator delete` ist in Vererbungshierarchien hilfreich, da hier der referenzierte Objekttyp nicht eindeutig ist. Er wird vom Compiler mit der Größe in Bytes des Objekts bzw. des Feldes initialisiert. Rückgabedatentyp von `operator delete` ist `void`.

Neben den bisher betrachteten Expressions `new` und `delete` gibt es noch die Gruppe der `placement new` Expressions. Diese zeichnen sich durch einen oder mehrere Parameter aus. Die in der C++-Standardbibliothek im Header `<new>` spezifizierte `placement new` Expression ist für die Konstruktion von Objekten an bestimmten, bereits allokierten, Speicherstellen gedacht.

```
new(Adresse) typ;
```

Der Einsatz dieser Variante wird im nächsten Kapitel beschrieben. Die Schaffung einer maßgeschneiderten bzw. anwendungsspezifischen `placement new` Expression erfolgt durch Definition eines `operator new`, der mit genau den zusätzlichen Parametern ausgestattet ist, die der `placement new` Expression übergeben werden sollen. Um Speicherlecks bei Exceptions zu verhindern, muss ein `operator delete` mit entsprechender Signatur vorhanden sein. Für weitere Details zu diesem Thema wird auf [3] verwiesen.

Nun soll ein für NUMA-Systeme optimierter `operator new[]` vorgestellt werden, der der Klasse `D` (siehe Listing 3.1), einem Wrapper um `double`, angehört. Im Rahmen der Beispielanwendung ist der Funktionsumfang von `D` identisch mit dem von `double`, lediglich die automatische Initialisierung von `D`-Objekten bei Nutzung von `std::new` ist zu berücksichtigen.

Listing 3.1: Wrapperklasse um `double`

```
class D {
      double d;
public:
      D(double _d=0.0) throw() : d(_d) {}
      ~D() throw() {}
      inline D& operator=(double _d) throw() {
            d=_d; return *this;
      }
      inline D operator+(const D& o) throw() {
            return D(d+o.d);
      }
      inline D operator*(const D& o) throw() {
            return D(d*o.d);
      }
      void* operator new[](size_t) throw(std::bad_alloc);
      void operator delete[](void*) throw();
      [...]
};
```

Listing 3.2 zeigt eine Version des `operator new`, die First-Touch-Placement mit einer parallelen Schleife realisiert. Dabei wird jedes Byte des Speicherbereichs berührt, da mit dem verwendeten Compiler aufgrund eines „Bugs“ nur Schleifen mit Stride 1 parallelisiert werden. Effizienter wäre eine Schleife, die nur die Startadressen der Objekte durchläuft, also die Erhöhung des Schleifenzählers um `sizeof(Typ)` für jede Iteration. Abhängig vom Compiler verwendet die `new` Expression den angeforderten Speicher auch zur Ablage von Verwaltungsinformation (s.o.). Um diese nicht zu überschreiben, darf das Datenfeld nicht wie bisher mit einer Konstanten initialisiert werden. Im Beispielcode ist ein nicht-veränderndes First Touch mit einer parallelen Schleife, in der jedes Byte gelesen und wieder geschrieben wird, umgesetzt. Da Speicher im überladenen `operator new[]` mittels `malloc()` allokiert wird, muss ein `operator delete[]` mit `free()` als Deallokationsfunktion bereitgestellt werden.

Der für die Klasse `D` entwickelte `operator new[]` wurde mit dem Vektortriaden-Benchmark auf Funktionsfähigkeit überprüft. Gegenüber einem `double`-Array muss bei einem Feld aus `D`-Objekten ein deutlicher Performancenachteil hingenommen werden. Dies kann der höheren Code-Komplexität z.B. der überladenen Rechenoperatoren zugeschrieben werden. Die Intention des überladenen `operator new[]` ist jedoch, korrektes NUMA-Placement auch für Arrays aus C++-Objekten zu ermöglichen. Für die einfache `double`-Wrapperklasse wurde dies erfolgreich umgesetzt, wie die Skalierungsstudie in Abbildung 3.2 zeigt. Im Vergleich zum 1-Thread Lauf (grüne Kurve) bringt die Verwendung von 2 Threads (braune Kurve) im Speicher doppelte und von 4 Threads (rote Kurve) vierfache Performance. Die 2 und 4-Thread Läufe mit dem Standard `operator new[]` können den Speicherbus besser auslasten als der entsprechende 1-Thread und sind so trotz nichtlokaler Zugriffe ca. 20 % schneller.

Für dynamische Objekte wie z.B. `vector<>` ist der gezeigte Placement-Ansatz nicht ausrei-

Listing 3.2: NUMA-optimierter `operator new[]` und zugehöriger `operator delete[]`

```
void* D::operator new[](size_t n) throw(std::bad_alloc) {
      void *m;
      if(!(m = malloc(n)) throw(std::bad_alloc);
      char *p = static_cast<char*>(m);
#pragma omp parallel for schedule(static)
      for(int i=0; i < n ; ++i){
            char a = p[i];
            p[i]=a;
      }
      return m;
}

void D::operator delete[](void* p) throw() {
      free(p);
}
```

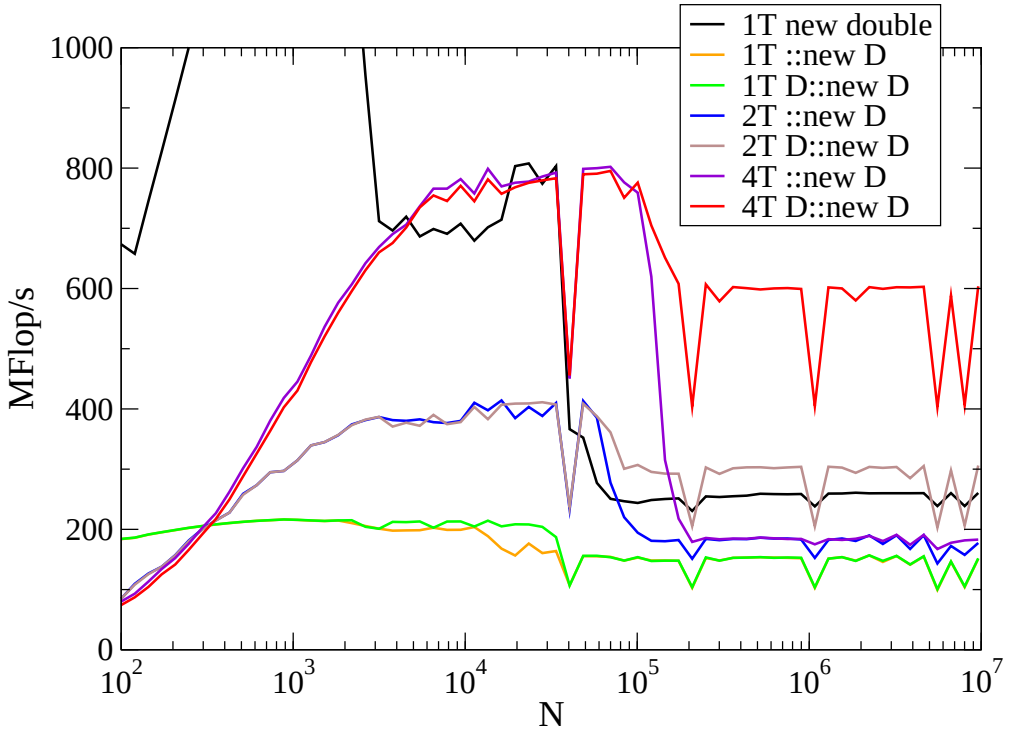

Abbildung 3.2: Performance der Vektortriade (Opteron) für Arrays aus C++-Objekten vom Typ `D`, einer einfachen Wrapperklasse um `double`.

chend. Wird ein Array aus Elementen vom Typ `vector<>` mit einem `operator new[]` wie dem in Listing 3.2 angelegt, so sind zwar die statischen Vektordaten Größe, Kapazität und der Zeiger auf den Speicherbereich für die Vektorelemente richtig auf die Lokalitätsdomänen verteilt, nicht jedoch die performancekritische Nutzlast. Korrektes Placement derartiger Datenstrukturen wird in den folgenden beiden Abschnitten behandelt.

3.2 STL-Vektor

In diesem Abschnitt wird beschrieben, wie STL-Objekte für den Einsatz in HPC-Anwendungen auf ccNUMA-Systemen optimiert werden können. Ein Vorteil für den Benutzer von STL-Containern ist die automatische Speicherverwaltung. Allokieren und Deallokieren von Speicher durch den Benutzer ist nicht nötig, was *Speicherlecks* verhindert. Containerelemente werden bei der Instanziierung initialisiert. Bei Containern wie `vector<>` kann die Größe mit Funktionen wie `resize()` und `pushback()` dynamisch verändert werden. Ein beim Einfügen zusätzlicher Elemente möglicherweise notwendiges Umkopieren der Daten in einen größeren Speicherbereich erfolgt automatisch und erfordert kein Eingreifen des Benutzers. Außerdem bieten STL-Container die Möglichkeit, beim Zugriff auf Containerelemente per Indexoperator die Indizes auf Gültigkeit zu überprüfen. Stellvertretend wird hier die Klasse `vector<>` behandelt, die einen komfortablen Ersatz für Arrays darstellt.

Gerade die dynamische Speicherverwaltung von STL-Containern macht bei der Umsetzung des NUMA-Placement tiefere Eingriffe erforderlich. Aufgrund der automatischen Initialisierung der Elemente ist eine einfache parallele Schleife nicht ausreichend. Eine (High-Level)-Allokationsfunktion wie z.B. `new` ist nicht vorhanden, kann also auch nicht wie in Abschnitt 3.1 ersetzt werden. Glücklicherweise stellen STL-Container eine Schnittstelle zur Verfügung, die dafür vorgesehen ist, Einfluss auf ihre interne Speicherverwaltung zu nehmen. Neben dem Typ-Parameter gibt es ein weiteres Template-Argument mit dem ein *Allocator* spezifiziert werden kann. Die vollständige Definition eines `vector<>`-Objekts sieht damit folgendermaßen aus:

```
vector<Typ, Allokator<Typ> > myvector(Groesse);
```

Ein Allokator ist wiederum eine Templateklasse, in ihr ist das Speichermanagement für den Container gekapselt. Ohne Angabe eines Allokators wird bei der Instanziierung eines Containers der Standardallokator `std::allocator<>` verwendet. Dieser ist Bestandteil der STL-Implementierung[1] und ist im Header `<memory>` [18] definiert. Modifikationen sind hier nicht möglich, also muss bei besonderen Anforderungen an den Allokator eine angepasste Allokatorklasse verwendet werden. Im Folgenden wird die Implementierung einer NUMA-optimierten Allokatorklasse beschrieben.

An eine für STL-Container geeignete Allokatorklasse werden einige Anforderungen gestellt, zum Beispiel bzgl. ihrer öffentlichen Schnittstelle. Diese muss dem ISO C++ Standard,

[1] STL-Bibliotheken gibt es z.B. von HP und SGI, aber auch die freie GNU Compiler Collection (GCC) ist damit ausgestattet.

Abschnitt 20.4.1 (siehe z.B. [19]) entsprechen. Hier sind neben Typdeklarationen auch die bereitzustellenden Methoden festgelegt. Davon werden die vier zentralen Funktionen genauer beleuchtet:

- Die Funktion `allocate()` ruft der Container auf, um Speicher zu allokieren. Übergeben muss er die Anzahl der Objekte, für die er Speicher anfordert. Der zweite Parameter ist optional und wird weder hier noch beim Standardallokator genutzt. Er ist dazu gedacht, einen Hinweis zur Lage des Speichers an den Allokator zu übergeben. Der Rückgabewert von `allocate()` ist ein Zeiger auf das erste Element in einem Speicherbereich der Größe `n * sizeof(T)`. `malloc()` garantiert die korrekte Ausrichtung des Speichers.
- Für jedes Objekt ruft der Container die Methode `construct()` auf. Neben einem Initialwert für das Objekt muss dessen Speicheradresse übergeben werden. Mit einer `placement new`-Expression (siehe S. 28) wird das Objekt an der spezifizierten Adresse erzeugt.
- Bevor der Container Speicher wieder freigeben kann, muss er die darin enthaltenen Objekte zerstören. Analog zur Objektkonstruktion muss er für jedes Objekt die Funktion `destroy()` aufrufen und die Objektadresse übergeben.
- Schließlich kann mit `deallocate()` Speicher freigegeben werden. Neben dem Zeiger auf den freizugebenden Speicherbereich hat auch `deallocate()` einen optionalen Parameter `n`, mit dem die Anzahl von Elementen spezifiziert werden kann, deren Speicher, beginnend mit dem ersten Element, zu deallokieren ist. Allerdings muss dieser den gleichen Wert haben wie das `n` beim Anlegen des Speichers. Die Übergabe eines Nullzeigers führt nicht zu einer Fehlermeldung[2], es wird jedoch auch kein Speicher freigegeben.

Auf den ersten Blick scheint die Funktion `construct()` und die darin enthaltene `placement new`-Expression der richtige Ansatzpunkt zur Umsetzung von NUMA-Placement zu sein. Der von `allocate()` allokierte Speicher ist ja noch unberührt, also keiner Lokalitätsdomäne zugeordnet. Jedoch wird die Adresse, an der ein Objekt konstruiert werden soll, vom Container vorgegeben, und ist bei der Nutzung von Standard-Containern auch nicht beeinflussbar. Deshalb muss für korrektes NUMA-Placement bereits in der Funktion `allocate()` gesorgt werden. Analog zum Vorgehen in den bisherigen Fällen wird eine parallele Initialisierungsschleife verwendet. Da `p` auf typlosen Speicher zeigt, wird er in einen Zeiger auf `char` umkopiert, mit dem in der Schleife jedes Byte initialisiert werden kann. In Grenzfällen liegen deshalb möglicherweise Objekte, die größer als 8 Byte sind, auf zwei physikalischen Seiten.

Listing 3.3 zeigt einen Ausschnitt aus der Allokatorklasse `malloc_allocator<>`, die von M. Austern vorgestellt wurde [20]. Die Funktion `allocate()` wurde um eine parallele Initialisierungsschleife erweitert.

In Abbildung 3.3 wird die Wirkung des optimierten Allokators anhand des Vektortriaden-Benchmarks verdeutlicht. Die 1-Thread Performance von `vector<>` mit Standardallokator ist identisch zu der von `vector<>` mit `numa_allocator<>`. Die Verwendung eines anderen Allokators als dem Defaultallokator bringt also keinen nachteiligen Overhead. Allerdings muss

[2] free(NULL) ist NOP in C

Listing 3.3: Auszug aus numa_allocator<>

```
typedef T                 value_type;
typedef value_type*       pointer;
typedef const value_type* const_pointer;
typedef std::size_t       size_type;

pointer allocate(size_type n, const_pointer = 0) {
      void* p = std::malloc(n * sizeof(T));
      char *m = static_cast<char*>(p);
#pragma omp parallel for schedule(static)
      for(int i=0; i < n*sizeof(T) ; i++){
            m[i]=0;
      }
      if (!p)
            throw std::bad_alloc();
      return static_cast<pointer>(p);
}
void deallocate(pointer p, size_type) {
      std::free(p);
}
void construct(pointer p, const value_type& x) {
      new(p) value_type(x);
}
void destroy(pointer p) {
      p->~value_type();
}
```

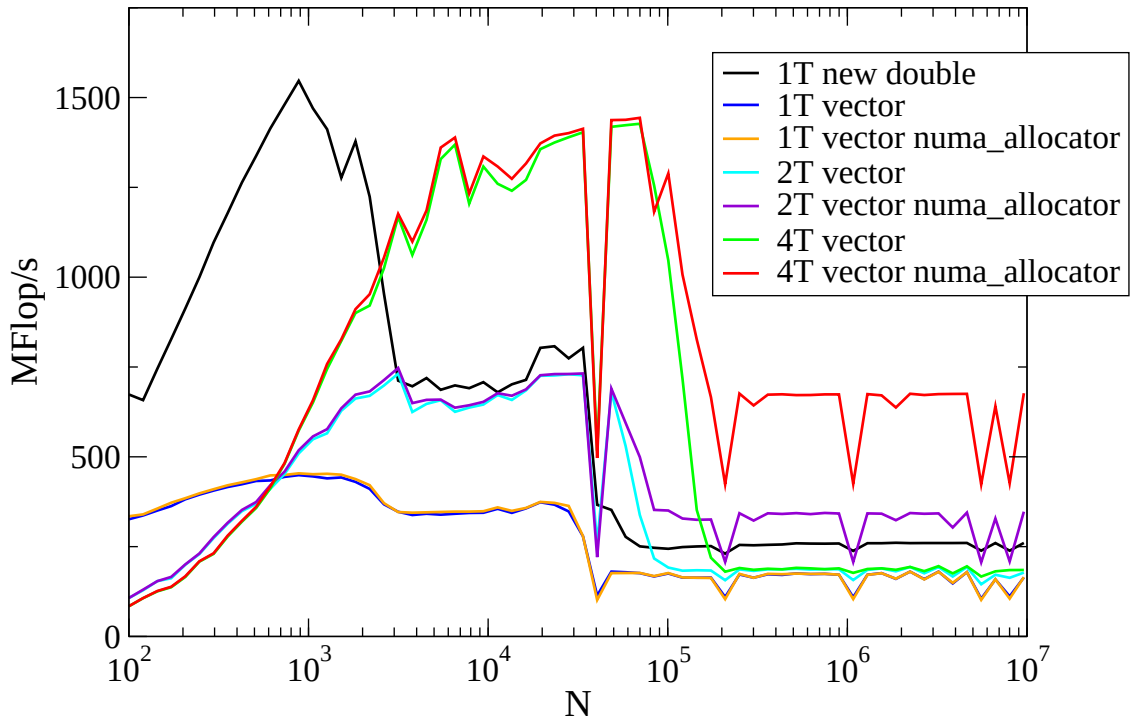

Abbildung 3.3: Performance der Vektortriade für STL-`vector<>`.

gegenüber einem `double`-Array ein Performanceverlust von etwa 30 % hingenommen werden. Die Skalierbarkeit ist mit dem `numa_allocator<>` jedoch nahezu perfekt. Zum Zugriff auf Container-Elemente in der Arbeitsschleife der Vektortriade wurde nicht wie bisher der Indexoperator verwendet. Das `i`-te Element von `vector<> a` kann statt mit `a[i]` effizienter mit

```
T::iterator ai = a.begin();
T element = *(ai + i);
```

dereferenziert werden. Dies entspricht der internen Arbeitsweise des Indexoperators, jedoch erzeugt dieser bei *jedem* Elementzugriff einen Iterator auf den Anfang des Containers. Gerade in Schleifen, in denen große Datenströme Element für Element durchlaufen werden, empfiehlt es sich, einen Iterator einmalig *vor* der Schleife anzulegen und als Basisadresse zu verwenden. Dadurch verdoppelt sich die Performance im Cache und im Speicher können etwa 20 % Geschwindigkeitszuwachs gemessen werden.

In Abschnitt 2.3 auf Seite 23 wurde die Möglichkeit beschrieben, die OpenMP-Blockgröße (`chunk_size`, CS) explizit festzulegen. Ihr Einfluss auf die Performance gilt für die verschiedenen „Vektor“-Container ebenso wie für Arrays. Dem Anwender des `numa_allocator<>` muss es also ermöglicht werden, die Blockgröße zu spezifizieren. Diese muss zum Zeitpunkt des Compilierens bekannt sein, deshalb empfiehlt sich die Einführung eines weiteren Templateparameters [16].

```
template <class T, int CS = 0> class numa_allocator{
```

```
...
#pragma omp parallel for schedule(static, CS)
...
};
```

Die ursprüngliche Version ohne Spezifizierung der Blockgröße muss weiterhin erreichbar sein, weshalb das Template für diesen Fall zu spezialisieren ist.

```
template <class T> class numa_allocator<T,0>{
...
#pragma omp parallel for schedule(static)
...
};
```

Aus den genannten Gründen sollte der Speicher in der generischen Variante an Speicherseiten ausgerichtet angelegt werden.

Das Allokatorkonzept bietet ein mächtiges Mittel zur Einflussnahme auf die Speicherverwaltung von STL-Containern. Allerdings treten bei der Kombination verschiedener Allokatoren mit dem gleichen Containertyp Probleme auf, da sich durch die Verwendung eines Allokators der Datentyp des Containers ändert. Zum Beispiel schlägt die Zuweisung eines `vector<>`-Containers mit Standardallokator an einen mit `numa_allocator<>` fehl. Im Allgemeinen ist ein passender Zuweisungsoperator nicht definiert. Diese und andere Mängel sollen der im nächsten Abschnitt beschriebenen Entwicklung eines Containers beseitigt werden.

3.3 NUMA-Vektor

Der Container `vector<>` aus der STL ist angesichts seines Funktionsumfangs erste Wahl, wenn es um bequeme und flexible Verwaltung von Objekten geht. Er entspricht damit aber eher einem klassischen Array als einem Vektor aus der Mathematik. Gerade im wissenschaftlichen Rechnen, bei der Umsetzung mathematischer Modelle, sind Vektoren allgegenwärtig. Mit dem `valarray<>` aus der C++-Standardbibliothek scheint hierfür ein geeigneter Datentyp gegeben zu sein. Dieser bietet z.B. elementweise Addition zweier Vektoren oder eine Maximumfunktion, Operationen, die üblicherweise den Einsatz fehleranfälliger Schleifen erfordern. Gegenüber dem STL-`vector<>` hat `valarray<>` jedoch den Nachteil, dass es keinen Iterator definiert hat. Die Grundvoraussetzung für die Nutzung von STL-Algorithmen ist also nicht erfüllt. Ein weiteres Manko ist das Fehlen eines Allokators. Für parallele Anwendungen auf ccNUMA-Systemen ist `valarray<>` damit nicht zweckmäßig.

Mit diesem Hintergrund wurde im Rahmen der Diplomarbeit ein Container entwickelt, der sowohl Allokator- und Iteratorkonzept der STL unterstützt, als auch vektorspezifische Funktionen anbietet. Aufgrund seines geplanten Einsatzbereichs wird er `numa_vector<>` genannt. Als Basis dient eine einfache Vektorklasse wie sie von I. Pohl vorgestellt wird [21]. Wie auch beim STL-`vector<>` kann der Benutzer über zwei Template-Argumente den Datentyp für die Vektorelemente und den zu verwendenden Allokator spezifizieren. Letzterer muss der

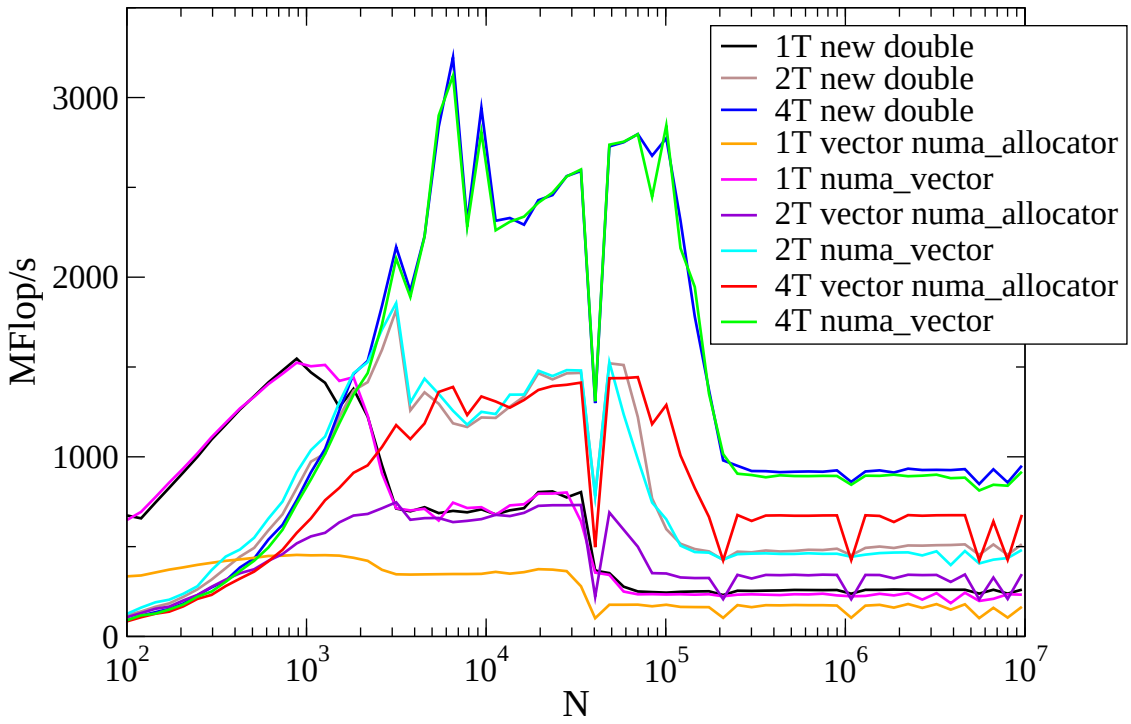

Abbildung 3.4: Performance der Vektortriade (Opteron) für `numa_vector<>`

STL-Spezifikation genügen, neben dem vorgestellten `numa_allocator<>` ist also auch der Defaultallocator der STL geeignet.

In Listing 3.4 ist das Gerüst von `numa_vector<>` abgebildet. Als Standardallokator kommt `numa_allocator<>` zum Einsatz. Neben den üblichen Konstruktorvarianten und einem Destruktor (die Implementierung des Kopierkonstruktors und des Destruktors sind in Listing 3.5 vollständig dargestellt) sind auch die vom STL-`vector<>` bekannten Funktionen `begin()`, `end()`, `size()` und `resize()` vorhanden. Letztere bedarf besonderer Aufmerksamkeit, da bei Größenänderung eines `numa_vector<>` das Placement angepasst werden muss (siehe Listing 3.6). Ebenfalls aus diesem Grund wurde auf die beim STL-`vector<>` praktizierte Allokation von Reservespeicher (`capacity`) und die Funktion `pushback()` verzichtet. Sowohl der Zuweisungsoperator als auch die ausgewählten arithmetischen Operatoren unterstützen Operanden, die mit unterschiedlichen Allokatoren instanziiert wurden. Implementierungsdetails sind Listing 3.7 zu entnehmen.

Ein weiterer Vorteil des selbst entwickelten Containers gegenüber `vector<>` ist, dass die Implementierung der Schleife, die mittels placement `new()` im allokierten Speicher Objekte ablegt, zugänglich ist. Insbesondere kann diese Schleife per OpenMP parallelisiert werden, was bei Objekten mit dynamischen Daten zu einem korrekten Placement der Speicherseiten auf dem Heap führt. Sowohl der überladene `operator new[]` als auch `vector<T, numa_allocator<T> >` lassen diese Optimierung nicht zu.

In Abbildung 3.4 wird die Performance der drei Container `numa_vector<>`, STL-`vector<>` mit `numa_allocator<>` und `double`-Array verglichen. Im Speicher ist `numa_vector<>` stets ca. 30 % besser als der STL-`vector<>` mit `numa_allocator<>`. Dies ist z.B. auf die sehr

Listing 3.4: numa_vector<>

```
template<class T, class Allocator = numa_allocator<T> > class numa_vector {
public:
      typedef T* iterator;
      explicit numa_vector(int=1);
      numa_vector(const T a[], int n);
      numa_vector(const numa_vector&);
      numa_vector(const iterator, const iterator);
      ~numa_vector();

      iterator begin() const { return p; }
      iterator end() const { return p+len; }
      int size() const { return len; }
      void resize(unsigned int);
      T& operator[](int);

      numa_vector& operator=(const numa_vector&);
      template<class OTHER_ALLOCATOR>
      numa_vector& operator=(const numa_vector<T,OTHER_ALLOCATOR>&);

      numa_vector& operator+=(const numa_vector&);
      template<class OTHER_ALLOCATOR>
      numa_vector& operator+=(const numa_vector<T,OTHER_ALLOCATOR>&);

      numa_vector operator+(const numa_vector&);
      template<class OTHER_ALLOCATOR>
      numa_vector operator+(const numa_vector<T,OTHER_ALLOCATOR>&);

      T operator*(const numa_vector&);
      template<class OTHER_ALLOCATOR>
      T operator*(const numa_vector<T,OTHER_ALLOCATOR>&);
private:
      T* p;
      unsigned int len;
};
```

Listing 3.5: Kopierkonstruktor und Destruktor für `numa_vector<>`

```
template<class T, class Allocator>
numa_vector<T,Allocator>::numa_vector(const numa_vector& v) {
      len=v.len;
      Allocator x;
      p = x.allocate(len);  // allocate and place memory
      assert(p!=0);
      int i=0;
      iterator vi = v.begin();
      for( i; i<len; vi++, i++)
            x.construct(&p[i], *vi);
}
template<class T, class Allocator>
numa_vector<T,Allocator>::~numa_vector() {
      Allocator x;
      for(int i=len-1; i>=0; --i)
            x.destroy(&p[i]);
      x.deallocate(p,0);
}
```

einfache Implementierung des Iterators (`typedef T* iterator;`) und des Indexoperators (i.W. `return p[i];`) zurückzuführen. Aufgrund des schlanken Überbaus kann der Code vom Compiler sehr gut optimiert werden. Die Performance unterscheidet sich in keinem der gemessenen Fälle signifikant vom `double`-Array Referenzwert.

3.4 Segmentierte Datenstrukturen

Der Speicher von Mehrprozessorsystemen ist natürlicherweise segmentiert. Bei ccNUMA-Systemen ist er in Lokalitätsdomänen aufgeteilt. Die kleinstmögliche Speicherverwaltungseinheit ist in diesem Zusammenhang eine Speicherseite. Eine Speicherseite befindet sich in genau einer Lokalitätsdomäne. Trotz sorgfältigem Page Placement auf Applikationsebene kann es vorkommen, dass sich von einem Thread benötigte Daten nicht in der eigenen Lokalitätsdomäne befinden und teure nichtlokale Speicherzugriffe erforderlich sind. Dies passiert genau dann, wenn eine Speicherseite von mehreren Threads genutzt wird. Nur der Thread, der die Seite zuerst berührt, kann lokal auf sie zugreifen. Falls dieser sie nicht komplett ausfüllt, kommt es zwangsläufig zu nichtlokalen Zugriffen durch andere Threads. Je größer die Speicherseiten in Relation zum Gesamtdatenvolumen pro Thread sind, desto größer ist der „Verschnitt“ und damit der negative Effekt.

Ein ähnliches Phänomen existiert bei Multicore UMA-Systemen auf Cacheebene. Die Kohärenzeinheit ist hier eine Cachezeile. Wenn der Inhalt einer Cachezeile fortlaufend von mehreren Threads verändert wird, muss diese zur Gewährleistung der Kohärenz der Caches aller Prozessoren ständig in den Speicher zurückgeschrieben und neu geladen werden. Der

Listing 3.6: `resize()` für `numa_vector<>`

```
template<class T, class Allocator>
void numa_vector<T,Allocator>::resize(unsigned int new_len){
      T *new_p;
      if(new_len < len){
            assert(new_len > 0);
            Allocator x;
            new_p = x.allocate(new_len);
            assert(new_p!=0);
            for(int i=0; i<new_len; i++)
                  x.construct(&new_p[i], p[i]);
            for(int i=0; i<len; i++)
                  x.destroy(&p[i]);
            x.deallocate(p,0);
            p = new_p;
            len = new_len;
      } else if(new_len > len){
            assert(new_len > 0);
            Allocator x;
            new_p = x.allocate(new_len);
            assert(new_p!=0);
            for(int i=0; i<len; i++)
                  x.construct(&new_p[i], p[i]);
            T def(0);
            for(int i=len; i<new_len; i++)
                  x.construct(&new_p[i], def);
            for(int i=0; i<len; i++)
                  x.destroy(&p[i]);
            x.deallocate(p,0);
            p = new_p;
            len = new_len;
      }
}
```

Listing 3.7: Skalarprodukt für `numa_vector<>`

```
template<class T, class Allocator>
T numa_vector<T,Allocator>::operator*(const numa_vector& v){
      assert(v.len == len);
      T temp;
      for(int i=0; i<len; i++)
            temp += p[i] * v.p[i];
      return temp;
}
template<class T, class Allocator> template<class OTHER_ALLOCATOR>
T numa_vector<T,Allocator>::operator*(const
                              numa_vector<T,OTHER_ALLOCATOR>& v){
      assert(v.size() == len);
      T temp;
      iterator me = begin();
      numa_vector<T,OTHER_ALLOCATOR>::iterator you = v.begin();
      for(int i=0; i<len; ++i)
            temp += *(me+i) * *(you+i);
      return temp;
}
```

durch dieses sogenannte *False Sharing* erzeugte Datenverkehr kann die Applikation sehr stark bremsen [6].

Das Problem ist in beiden Fällen nichtexklusiver Zugriff auf eine Speicherseite. Um jedem Thread exklusiven Zugriff auf eine Speicherseite zu ermöglichen, müssen die Speicherstellen, die zeitlich lokal von verschiedenen Threads verändert werden, z.B. durch Einbau von Pufferspeicher (Padding), räumlich getrennt werden. Um diese Strategie beherrschbar zu machen, muss sich die Segmentierung des Speichers im verwendeten Datentyp manifestieren. Im Folgenden (Listing 3.9) wird ein Container vorgestellt, dessen Datenhaltung explizit segmentiert ist. Der logische Aufbau ist in Abbildung 3.5 skizziert. Containerelemente sind in Segmenten gespeichert, die über eine Segmentliste adressiert werden können. Für den Zugriff auf ein Element ist also die Segmentadresse und die Adresse innerhalb des Segments erforderlich. Umgesetzt wurde diese zweidimensionale Struktur mit einem STL-`vector<> nodes`, dessen Elemente Verweise auf die Segmente sind. Die Segmente werden sequenziell in einem Array gespeichert. Der Abstand zweier benachbarter Segmente kann mittels eines Padding-Parameters festgelegt werden (siehe Abbildung 3.6). Jedes Segment enthält grundsätzlich gleich viele Elemente (`columns`). Verschnitt wird auf die ersten `size % segments` Segmente verteilt. Ein Segment wird von der Klasse `segm<>` (Listing 3.8) repräsentiert.

Um alle Containerelemente, z.B. zur Initialisierung, durchlaufen zu können, sind zwei geschachtelte Schleifen (eine über die Segmente und eine über das aktuelle Segment) erforderlich. Für den Zugriff auf einzelne Elemente kann ein Klammeroperator implementiert werden. Die allgemeine Schnittstelle für den Umgang mit Containerelementen ist ein Iterator. Im folgenden Abschnitt werden die Besonderheiten von Iteratoren im Zusammenhang

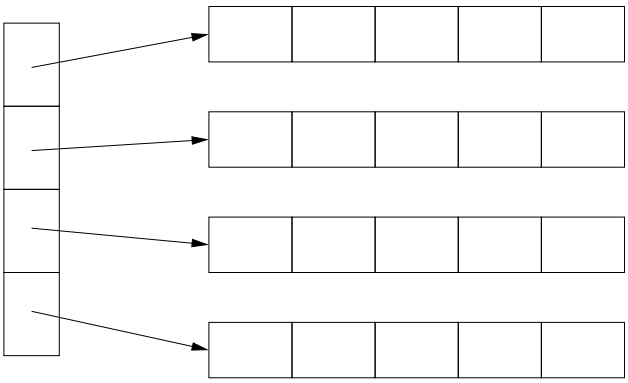

Abbildung 3.5: Segementierte Datenstruktur

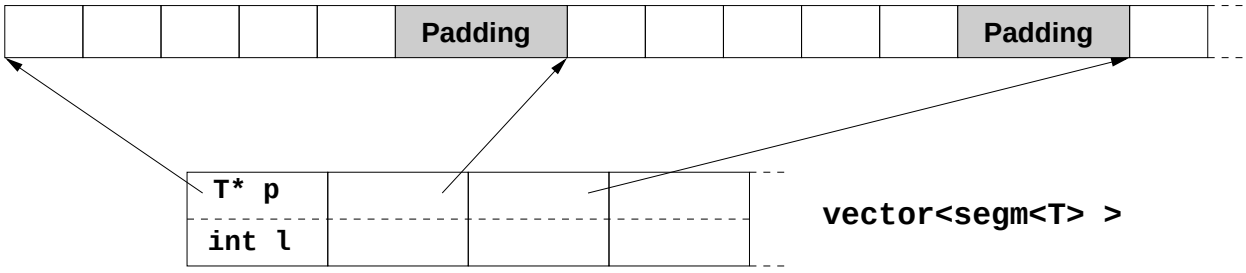

Abbildung 3.6: Datenhaltung von `seg_array`

Listing 3.8: Segmentklasse `segm<>`

```
template<typename T> class segm{
    friend class seg_array;
public:
    segm(T* addr=0, int cols=0):p(addr),l(cols){}
    T* begin(){ return p; }
    T* end(){ return p+l; }
private:
    T* p;
    int l;
};
```

Listing 3.9: Segmentierte Datenstruktur `seg_array<>`

```
template <class T>
class seg_array {
public:
    template<typename T> class segm;
    typedef seg_array_iter<T> iterator;
    typedef typename vector<segm<T> >::iterator segment_iterator;
    typedef T* local_iterator;
    seg_array(int segments=1, int size=1, int padding=0) {
        int columns = size/segments;
        int missing_columns = size%segments;
        nodes.resize(segments+1);
        data = static_cast<char*>(malloc(size*sizeof(T) +
                                                (segments-1)*padding));
        for(int i=0; i<missing_columns; ++i){
            nodes[i].p = reinterpret_cast<T*>
                (data + i * ((columns+1)*sizeof(T) + padding));
            nodes[i].l = columns+1;
        }
        for(int i=missing_columns; i<nodes.size()-1; ++i){
            nodes[i].p = reinterpret_cast<T*>
            (data + missing_columns*sizeof(T) + i *
                                    (columns*sizeof(T) + padding));
            nodes[i].l = columns;
        }
        nodes[nodes.size()-1].p = 0;
        nodes[nodes.size()-1].l = 0;
        int n_segments = nodes.size()-1;
#pragma omp parallel for schedule(static)
        for(int i=0; i<n_segments; ++i){
            char *m = reinterpret_cast<char*>(nodes[i].p);
            for(int k=0; k<(sizeof(T) * nodes[i].l); ++k){
                m[k]=0;
            }
            for(int j=0; j<nodes[i].l; ++j){
                new((nodes[i].p)+j) T();
            }
        }
    }
    ...
private:
    char* data;
    vector<segm<T> > nodes;
};
```

Listing 3.10: Iterator für `seg_array`

```
template <class T> class seg_array_iter {
      friend class seg_array<T>;
public:
      seg_array_iter(T* localit, typename vector<seg_array<T>::segm<T> >
            ::iterator nodeit) : cur(localit), node(nodeit) {}
      seg_array_iter& operator++() {
            if(++cur == (**node).end() && (*(node+1))) {
                  ++node;
                  cur = (**node).begin();
            }
            return *this;
      }
      ...
private:
      T* cur;
      typename vector<seg_array<T>::segm<T> >::iterator node;
};
```

mit segmentierten Datenstrukturen behandelt (siehe [24]).

3.5 Segmentierte Iteratoren

Iteratoren sind der Mechanismus, der es ermöglicht, Algorithmen von Containern zu entkoppeln: STL-Algorithmen sind Templates und werden mit dem Iteratortyp parametrisiert, damit sind sie nicht auf einen einzigen Containertyp beschränkt [22]. Neben den von Zeigern bekannten trivialen Iteratoreigenschaften Dereferenzierbarkeit und Vergleichbarkeit, muss ein Iterator eine weitere Anforderung erfüllen. Operatoren zur Bestimmung von Vorgänger und Nachfolger (`--` und `++`) bzw. für Iteratorarithmetik (`+` und `-`) werden je nach Iteratorkategorie (s.u.) gefordert [23].

Beziehungen zu Vorgängern und Nachfolgern sind bei linearen Datenstrukturen direkt an Speicherstellen gekoppelt. Die Adresse eines Nachfolgers kann aus der eigenen Adresse und der Größe der verwalteten Objekte berechnet werden. Bei segmentierten Datenstrukturen ist dies nur innerhalb eines Segments möglich. Am Segmentende muss, falls vorhanden, auf das nächste Segment umgeschaltet werden. Um existierende Algorithmen, die üblicherweise auf gleichförmigen, linearen Datenstrukturen basieren, auch auf segmentierte Container anwenden zu können, erscheint es sinnvoll, die Segmentierung in den Operatoren zu kapseln. In Listing 3.10 wird ein einfacher vorwärts-Iterator (nach [24]) für `seg_array` vorgestellt, bei dem der `operator++` die Segmentierung des Containers transparent macht.

Damit ist es leicht möglich, `seg_array` als Container für den Vektortriaden-Benchmark zu nutzen. Der Rechenkern wurde modifiziert, so dass statt dem Indexoperator dort jetzt der

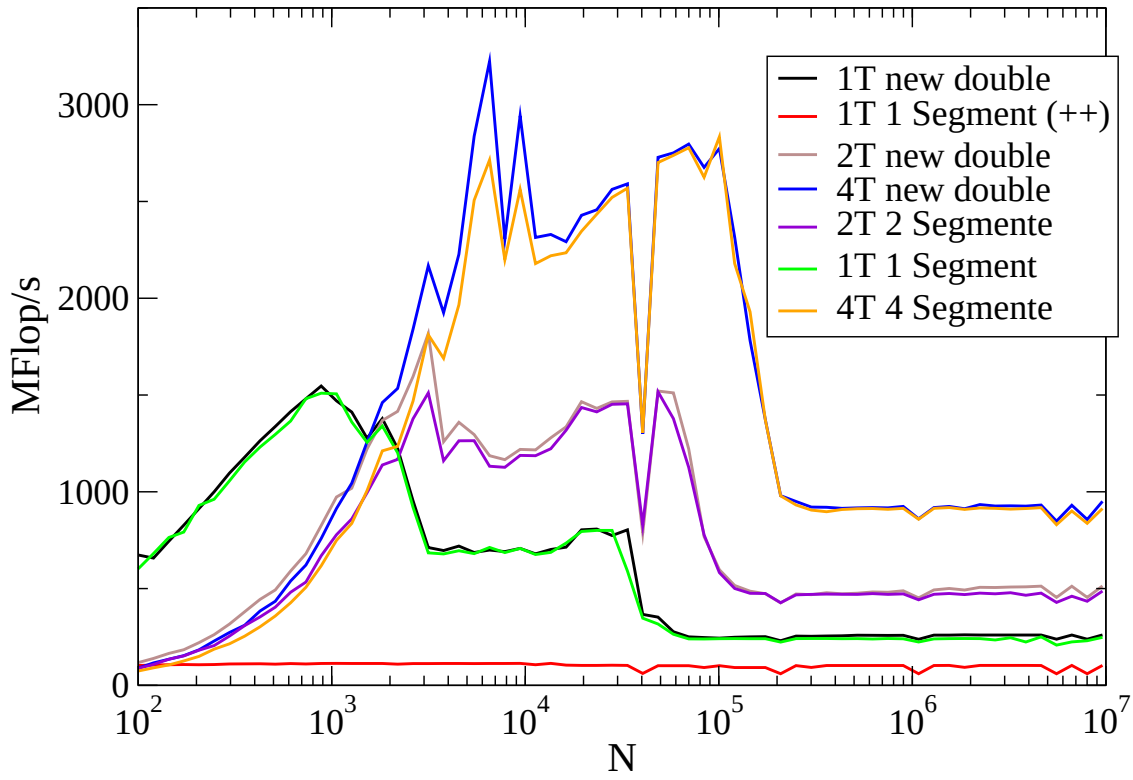

Abbildung 3.7: Performance der Vektortriade (Opteron) für `seg_array`

`operator++` von `seg_array_iter` zum Einsatz kommt. Die innerste Schleife hat dann die Form:

```
for(int i=0; i<N; ++i) {
      *ai = *bi + *ci * *di;
      ++ai, ++bi, ++ci, ++di;
}
```

Um Vergleiche mit den bisherigen Containern zu erleichtern, wurde `seg_array` mit einem Segment und damit ohne Padding verwendet. Jedes der vier Felder liegt damit effektiv in einem einfachen `double`-Array. Abbildung 3.7 zeigt die Messergebnisse. Mit knapp über 100 MFlop/s wurden selbst im Speicher nur 40 % des `new double` Referenzwerts erreicht. Der Grund für dieses inakzeptable Resultat ist im Iterator zu finden. Bei jedem Durchlauf der Rechenschleife werden alle vier Iteratoren inkrementiert. Dabei findet jedes mal die Überprüfung auf das Segmentende statt. Dieser Overhead ist für die Performanceprobleme verantwortlich.

Die Nutzung des naiv implementierten Iterators könnte selbstverständlich vermieden werden, indem die innerste Schleife der Vektortriade explizit an die segmentierte Datenstruktur angepasst wird:

```
for(int i=0; i<segments; ++i){
      for(int j=0; j<elements_per_segment; ++j){
```

```
            a(i,j) = b(i,j) + c(i,j) * d(i,j);
      }
}
```

Ein derartiger Umbau ist bei hochoptimierten numerischen Algorithmen unmöglich, falls sie z.B. in Bibliotheken ausgelagert sind, zumindest aber unerwünscht, da die Anpassung für jeden verwendeten Containertyp vorgenommen werden müsste. In C++ gibt es eine elegante Methode, mit der Segmentverwaltung und Rechnung ohne Performanceeinbußen getrennt werden werden kann.

Die zugrundeliegende Technik, eine Traits-Klasse, wird z.B. auch in den generischen STL-Algorithmen genutzt. Dort wird das Ziel verfolgt, für alle Iteratortypen mit einer einheitlichen Schnittstelle möglichst effizienten Code zur Verfügung zu stellen. STL-Algorithmen haben als einzigen Templateparameter den Typ der übergebenen Iteratoren. Iteratoren, die zentrale Abstraktion in der STL, sind in verschiedene Kategorien einteilbar, an die unterschiedlich hohe Anforderungen gestellt werden. Die drei wichtigsten werden hier — mit aufsteigendem Funktionsumfang — kurz vorgestellt.

- Ein einfacher *vorwärts-Iterator* muss z.B. die Operation `++` (Weiterschalten zum direkten Nachfolger) unterstützen.
- Möchte man mit dem Iterator nicht nur den Nachfolger, sondern auch den Vorgänger im Container erreichen (`--`), so muss dieser mindestens vom Typ *bidirektional* sein.
- Der mächtigste Iterator, der *random-access-Iterator*, bietet zusätzlich zu den genannten Funktionen z.B. die Möglichkeit, das n-te Element einer Folge direkt zu referenzieren und beherrscht wie ein normaler C-Pointer auch Zeigerarithmetik.

Nutzt ein Algorithmus diese exklusive Eigenschaft des random-access-Iterators, so ist er für die schwächeren Iteratorkategorien nicht geeignet. Umgekehrt funktioniert ein Algorithmus, der einen vorwärts-Iterator benötigt, auch mit einem bidirektionalen oder random-access-Iterator. Abhängig vom Iteratortyp können unterschiedlich effiziente Algorithmen formuliert werden. Beispielsweise ist ein Algorithmus `reverse<>()`, der zum Umkehren der Elementreihenfolge in einem Container einen bidirektionalen Iterator nutzt, schneller, als einer, der hierfür einen vorwärts-Iterator verwendet [24]. Der schnellere Algorithmus ist dann aber nicht mehr für den vorwärts-Iterator geeignet. Das Ziel ist ein Algorithmus, der sowohl universell, d.h. mit allen Iteratortypen verwendbar, als auch möglichst effizient ist.

Dazu wird ein *Dispatcher Algorithmus* mit identischem Interface eingeführt (s.u.). Er entscheidet anhand des Iteratortyps, welche Version des Codes ausgeführt werden kann. Der Iteratortyp ist allerdings kein Typ im Sinne eines Datentyps, sondern ein abstrakter Katalog von Anforderungen, die der Iterator erfüllt. Der C++-Überlademechanismus kann deshalb nicht direkt genutzt werden, um z.B. ein `reverse` für vorwärts-Iteratoren von einem für bidirektionale Iteratoren zu unterscheiden. Zu diesem Zweck werden sie mit einem weiteren Funktionsparameter (einem *Tag*) ausgestattet, der den Iteratortyp repräsentiert:

```
template <class Iter>
void reverse(Iter first, Iter last, forward_iterator_tag) {...}
```

```
template <class Iter>
void reverse(Iter first, Iter last, bidirectional_iterator_tag) {...}
```

Ein Tag ist eine leere Klasse, die als Etikett für einen Iteratortyp dient. Für jeden Iteratortyp, der unterstützt werden soll, muss eine Tag-Klasse und eine Spezialisierung des Algorithmus vorhanden sein. Der Dispatcher entnimmt aus einer Traits-Klasse den dort dem Iteratortyp zugeordneten Tag, um damit den geeigneten Algorithmus aufzurufen:

```
template <class Iter>
void reverse(Iter first, Iter last) {
      typedef typename iterator_traits<Iter>::iterator_category category;
      reverse(first, last, category());
}
```

Die Traits-Klasse für Iteratortypen `iterator_traits` ist in der C++-Standardbibliothek enthalten. Traits-Klassen können aber auch selbst erstellt werden, um generischen Algorithmen je nach Wert der Templateparameter spezifische Informationen und Funktionen zur Verfügung zu stellen. Es sind Algorithmen denkbar, die statt mit verschiedenen Iteratoren mit unterschiedlichen Containertypen umgehen können [25]. Da das „Dispatching“ bereits bei der Übersetzung stattfindet, entsteht dadurch kein Overhead zur Laufzeit.

Der Traits-Mechanismus kann auch zur Unterscheidung zwischen segmentierten und nichtsegmentierten Iteratoren herangezogen werden. In Listing 3.11 ist eine geeignete Traits-Klasse abgebildet, anhand derer der in Listing 3.12 gezeigte Dispatcher die mit `true_type` oder `false_type` markierte Triadenfunktion auswählen kann.

Das Konzept des segmentierten Iterators ist zu den im C++-Standard definierten Iteratortypen orthogonal. Ein segmentierter Iterator ist also z.B. ein bidirektionaler Iterator, jedoch mit der zusätzlichen Eigenschaft, auf einen segmentierten Container zu verweisen. Der Typ eines segmentierten Iterators ist gleich dem schwächsten Typ seiner beiden Komponenten. Diese sind ein Segmentiterator, der auf ein Segment zeigt, und ein lokaler Iterator, der auf ein Element innerhalb dieses Segments zeigt. Damit können z.B. die verschiedenen Implementierungen der Vektortriade (s.o.) mit einheitlichem Interface verfügbar gemacht werden, d.h. der Algorithmus kann erkennen, mit welchem Iteratortyp er aufgerufen wird. Falls es sich um einen nichtsegmentierten Iterator handelt, kann die effiziente Version ohne überladenen `operator++` verwendet werden (`false_type`). Der überladene `operator++` muss wegen seines negativen Effekts auch dann gemieden werden, wenn ein segmentierter Iterator genutzt wird. Deshalb wird (analog zu [24]) ein Ansatz verfolgt, der dies ermöglicht.

Die vier segmentierten Container, die die Vektoren enthalten, werden nicht am Stück, sondern Segment für Segment an die Vektortriaden-Funktion übergegeben. Dazu werden in der Traits-Klasse für einen segmentierten Iterator Funktionen bereitgestellt, mit denen aus dem segmentierten Iterator ein Segmentiterator und ein lokaler Iterator ermittelt werden kann. Für ersteren können wiederum zwei lokale Iteratoren auf Anfang und Ende des Segments abgefragt werden. Mit diesen nichtsegmentierten Iteratoren (`false_type`) kann die herkömmliche, schnelle Vektortriaden-Funktion aufgerufen werden. Es wird vorausgesetzt,

Listing 3.11: Traits-Klasse für segmentierten Iterator

```
template <class Iterator>
struct segmented_iterator_traits {
      typedef false_type is_segmented_iterator;
};

template <class T>
struct segmented_iterator_traits<seg_array_iter<T> > {
      typedef true_type is_segmented_iterator;
      typedef typename seg_array<T>::iterator iterator;
      typedef typename seg_array<T>::segment_iterator segment_iterator;
      typedef typename seg_array<T>::local_iterator local_iterator;

      static segment_iterator segment(iterator it) {
            return it.segment();
      }
      static local_iterator local(iterator it) {
            return it.local();
      }
      static local_iterator begin(segment_iterator sit) {
            return (*sit).begin();
      }
      static local_iterator end(segment_iterator sit) {
            return (*sit).end();
      }
};
```

Listing 3.12: Vektortriade mit Dispatcher

```
template <class T>
void do_segmented_array_iterator_triad(T, T, T, T, T, int);

template <class T>
void do_segmented_array_iterator_triad(T a, T b, T c, T d, T ae, int N,
            false_type) {...}  // Code f"ur nichtsegmentierten Iterator

template <class T>
void do_segmented_array_iterator_triad(T a, T b, T c, T d, T ae, int N,
            true_type) {...}  // Code f"ur segmentierten Iterator

template <class Iter>
void do_segmented_array_iterator_triad(Iter a, Iter b, Iter c,
                                       Iter d, Iter ae, int N){
      typedef segmented_iterator_traits<Iter> Traits;
      do_segmented_array_iterator_triad(a, b, c, d, ae, N,
                              typename Traits::is_segmented_iterator());
      return;
}
```

dass die vier Container gleich strukturiert sind, d.h. Anzahl und Größe der Segmente identisch sind. Unter der Annahme, dass die Segmente nicht zu kurz sind, ist die Vektortriade für segmentierte Datenstrukturen gleich performant wie für reine C-Arrays. Der Code-Zweig für segmentierte Iteratoren (`true_type`) dient damit nur zum „Entpacken“ der Segmente aus dem Container und zum rekursiven Aufruf der Vektortriaden-Funktion. In Listing 3.13 wird eine parallele Implementierung gezeigt.

Jeder Thread bekommt eine Anzahl von Segmenten zugewiesen. Im `seg_array<>`-Konstruktor (siehe Listing 3.9) werden die Segmente nach dem selben Muster in die Lokalitätsdomänen platziert. Für die Vektortriade, einem Algorithmus bei dem es keine Abhängigkeiten zwischen entfernten Elementen gibt, können damit durch geeignete Wahl der Segmentgröße und des Paddings nichtlokale Speicherzugriffe eliminiert werden. Die Vektortriaden-Performance eines `seg_array<>` mit einem Segment pro Thread (ohne Padding) weicht nur geringfügig vom Referenzwert ab (siehe Abbildung 3.7). Diese Differenz ist auf die höhere Komplexität des `seg_array<>` zurückzuführen.

Optimale Werte für Segmentanzahl und Padding hängen von zu vielen Faktoren ab, um sie theoretisch bestimmen zu können. Deshalb müssen sie für jeden Anwendungsfall experimentell ermittelt werden. Allerdings führt — bei konstanter Problemgröße — eine Erhöhung der Segmentanzahl zu einer Verkleinerung der einzelnen Segmente. Abbildung 3.8 zeigt, dass die Performance damit absinkt, weil Pipelining- und Prefetching-Mechanismen nicht mehr greifen [6]. Im gezeigten Fall sollte die Segmentlänge 20000 Elemente nicht unterschreiten. Typischerweise würde man die Zahl der Segmente gleich der Zahl der Threads wählen. Da Padding bytegenau festgelegt werden kann, muss die Ausrichtung der Segmente im Speicher

Listing 3.13: Vektortriade für segmentierte Iteratoren (`true_type`)

```
template <class Iter>
void triad(Iter a, Iter b, Iter c,
           Iter d, Iter ae, int N, true_type){
      typedef segmented_iterator_traits<Iter> traits;
      typename traits::segment_iterator afirstsegm = traits::segment(a);
      typename traits::segment_iterator alastsegm = traits::segment(ae);
      typename traits::segment_iterator bfirstsegm = traits::segment(b);
      typename traits::segment_iterator cfirstsegm = traits::segment(c);
      typename traits::segment_iterator dfirstsegm = traits::segment(d);

      int i, segments = alastsegm - afirstsegm + 1;
      typename traits::segment_iterator atmp;
      typename traits::segment_iterator btmp;
      typename traits::segment_iterator ctmp;
      typename traits::segment_iterator dtmp;
#pragma omp parallel for private(atmp,btmp,ctmp,dtmp)
      for(i=0; i<segments; ++i) {
            atmp = afirstsegm + i;
            btmp = bfirstsegm + i;
            ctmp = cfirstsegm + i;
            dtmp = dfirstsegm + i;
        triad(traits::begin(atmp), traits::begin(btmp), traits::begin(ctmp),
              traits::begin(dtmp), traits::end(atmp), int N);
      }
      return;
}
```

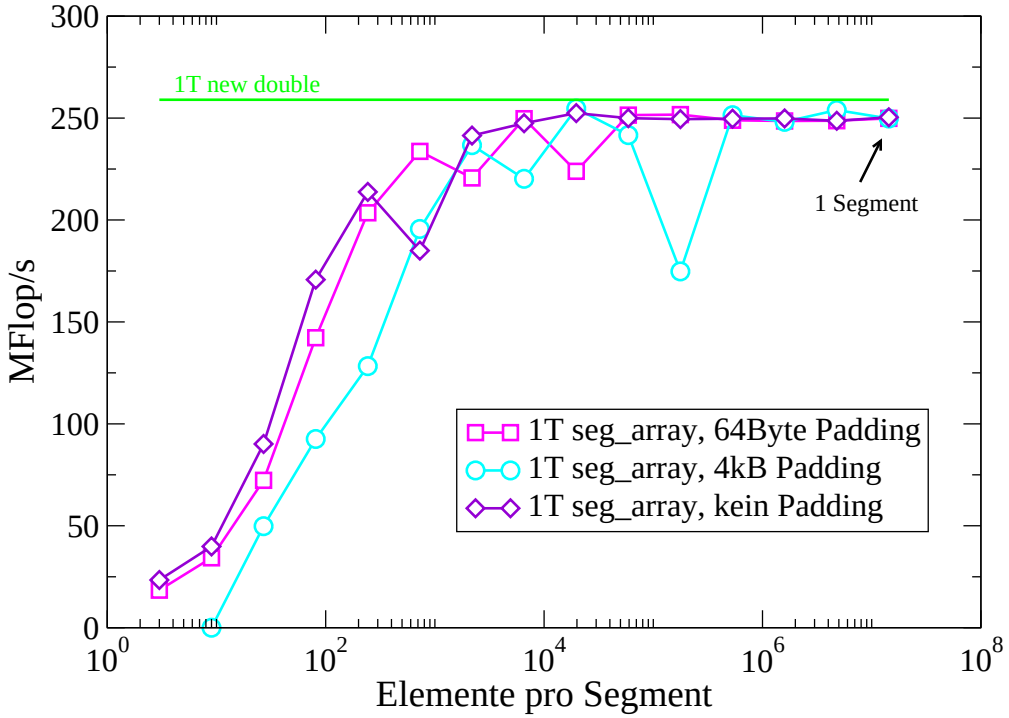

Abbildung 3.8: Performance der Vektortriade (Opteron) abhängig von der Anzahl der Segmente bei einer festen Feldgröße von 3^{15} Elementen.

beachtet werden. Segmentgröße und Padding sollten insgesamt Vielfache der Speicherseitengröße lang sein, einerseits um die in Abschnitt 2.3 beschriebenen Überlappungseffekte zwischen Speicherseiten zu umgehen, und andererseits um für Daten beliebigen Typs korrektes Alignment sicherzustellen.

Am Beispiel des `seg_array<>` wurde in diesem Abschnitt ein Container gezeigt, der an segmentierte Speicherstrukturen angepasst ist. Er eignet sich jedoch neben eindimensionalen Strukturen wie Arrays auch für mehrdimensionale, natürlicherweise segmentierte Datenstrukturen wie beispielsweise eine aus Zeilenvektoren aufgebaute Matrix. Diese werden z.B. bei dem im folgenden Kapitel beschriebenen Relaxationsverfahren genutzt.

4 Anwendung NUMA-optimierter Container für Relaxationsverfahren

Im letzten Kapitel wurde anhand des synthetischen Vektortriaden-Benchmarks gezeigt, wie NUMA-Placement für C++-Container realisiert werden kann. Diese Programmiertechniken sollen nun auf ihre Tauglichkeit in Applikationen überprüft werden. Als Testumgebung dient der in Abschnitt 1.6.2 beschriebene Relaxationsbenchmark. Die übliche Performancemetrik ist bei Anwendungen, die Berechnungen auf Gittern ausführen, die Einheit `MLup/s` (Mega Lattice Site Updates per Second = Millionen Gitterpunktaktualisierungen in der Sekunde). Gegenüber einer reinen Laufzeitmessung hat dies den Vorteil, dass die Ergebnisse für verschiedene Gittergrößen und -formen direkt vergleichbar sind. Aus der gemessenen Laufzeit t_{wall} wird die Performance mit folgender Formel berechnet:

$$P_{\text{relax}} = \frac{N_s \cdot N^d \text{ [Lup]}}{t_{\text{wall}} \text{ [s]}} \quad , \tag{4.1}$$

wobei N_s die Zahl der Sweeps, N die Größe des Simulationsgebietes in einer Dimension und d die Zahl der Dimensionen ist.

In jedem der folgenden Abschnitte werden die beiden beim Relaxationsbenchmark verwendeten Matrizen aus Zeilenvektoren verschiedenen Typs aufgebaut. Jeder Gitterpunkt wird durch einen `double` Wert repräsentiert, komplexe Objekte sind hier unüblich. Falls nicht anders angegeben, werden quadratische Gitter der Größe 8000 x 8000 Elemente verwendet und 300 Sweeps durchgeführt. Die Gebietszerlegung kann vertikal erfolgen, d.h. jeder Zeilenvektor wird auf die Threads verteilt, oder horizontal, dann bearbeitet jeder Thread einen Block ganzer Zeilen (siehe Abbildung 4.1).

4.1 C-Array

Eine Matrix kann in C++, wie in C und Fortran auch, aus einem Feld von Zeigern auf Felder aufgebaut werden:

```
double **matrix;
matrix = new double*[zeilen];
for(int i=0; i<zeilen; i++){
      matrix[i] = new double[spalten];
}
```

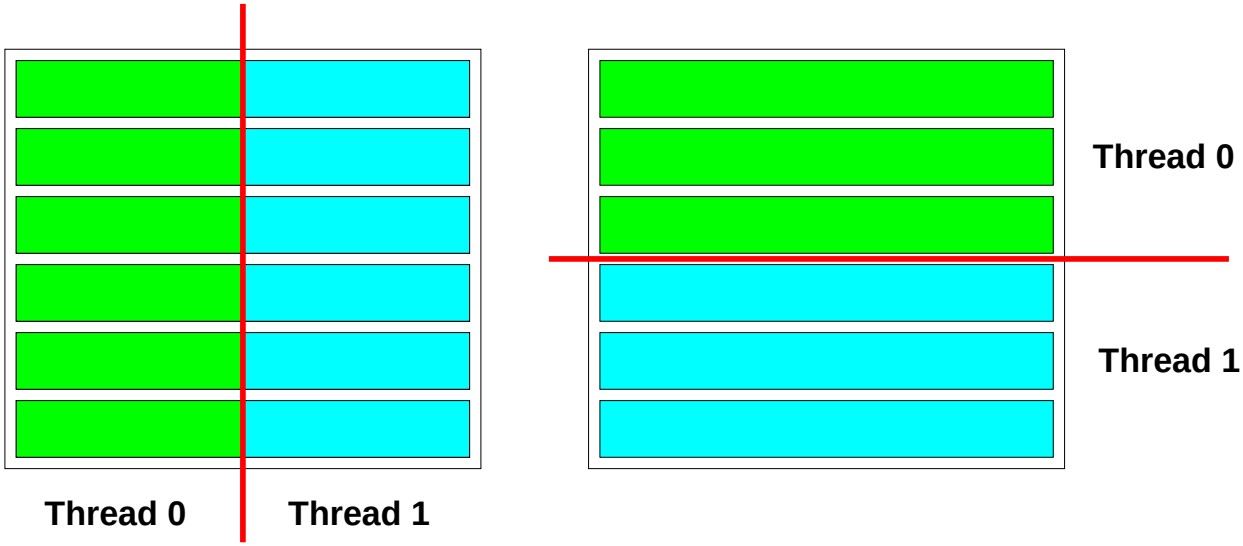

Abbildung 4.1: Gebietszerlegung für 2 Threads vertikal (links) und horizontal (rechts)

Für Basisdatentypen wie `double` findet das Placement bei der Feldinitialisierung statt. Vertikale Gebietszerlegung kann damit folgendermaßen umgesetzt werden:

```
for(int i=0; i<zeilen; i++){
#pragma omp parallel for schedule(static)
      for(int j=0; j<spalten; j++){
            matrix[i][j]=0.5;
      }
}
```

Nach dem gleichen Muster muss die Rechenroutine parallelisiert werden.

Horizontale Gebietszerlegung funktioniert analog, hier wird die Schleife über die Spalten parallelisiert. Außerdem müssen die Zeilenvektoren parallel angelegt werden, da die Performance sonst ca. 20 % schlechter ist als erwartet.

```
double **matrix;
matrix = new double*[zeilen];
#pragma omp parallel for schedule(static)
for(int i=0; i<zeilen; i++){
      matrix[i] = new double[spalten];
```

Dieser Unterschied tritt nicht bei der Messung mit einem Thread auf, was auf mangelhaftes NUMA-Placement hindeutet. Tatsächlich ist es so, dass in den für ein Array allokierten Speicherbereich bereits vor der Initialisierung Verwaltungsinformation (z.B. die Länge) geschrieben wird. Dadurch wird die Speicherseite, die diesen Bereich des Arrays enthält, in den Speicher des ausführenden Threads platziert. Geschieht dies seriell, so sind Zugriffe auf

einige dieser Seiten bei paralleler Rechnung nichtlokal, was bei Arrays, die wenige Speicherseiten umfassen, einen signifikanten Anteil nichtlokaler Zugriffe nach sich zieht. Beim Vektortriaden-Benchmark ist dieser Effekt nicht sichtbar, da hier für kleine `N` alle vier Vektoren im Cache gehalten werden können. Für große Arrays relativiert sich in beiden Fällen die negative Auswirkung.

4.2 STL-Vektor und NUMA-Vektor

Eine Matrix kann aus STL-Vektoren folgendermaßen erzeugt werden:

```
vector<vector<double> > matrix(zeilen);
```

Die Zeilenvektoren müssen mit `resize()` auf die gewünschte Länge gebracht werden. Wird dies in einer parallelen Schleife ausgeführt, so wird das Gebiet horizontal zerlegt. Jeder Thread ruft `resize()` für „seine“ Zeilen auf, wodurch diese in seinem lokalen Speicher angelegt werden.

```
#pragma omp parallel for schedule(static)
for(int i=0; i<zeilen; i++){
      matrix[i].resize(spalten);
}
```

Vertikale Gebietszerlegung kann unter Zuhilfenahme des `numa_allocator<>` erreicht werden. Dort wird in der vom Vektor aufgerufenen Funktion `allocate()` der Speicherbereich für die Elemente auf die Lokalitätsdomänen verteilt. Die Parallelisierung der Schleife um `resize()` entfällt.

```
vector<vector<double, numa_allocator<double> > > matrix(zeilen);
for(int i=0; i<zeilen; i++){
      matrix[i].resize(spalten);
}
```

Statt `vector<>` kann analog der in Abschnitt 3.3 vorgestellte NUMA-Vektor verwendet werden.

4.3 Segmentiertes Array

In Abschnitt 3.4 wurde der Container `seg_array<>` vorgestellt, der auf segmentierte Datenstrukturen ausgelegt ist. Eine aus Zeilenvektoren aufgebaute Matrix ist ein typischer Vertreter einer segmentierten Datenstruktur. Eine rechteckige Matrix kann damit folgendermaßen erzeugt werden:

```
int elemente = zeilen * spalten;
seg_array<double> matrix(zeilen,elemente,0);
```

Jeder Zeilenvektor wird in einem Segment des `seg_array<>` gespeichert. Die Matrix wird automatisch auf die Lokalitätsdomänen der beteiligten Threads aufgeteilt, da die Segmente in einer parallelen Region konstruiert werden. Vertikale Gebietszerlegung ist deshalb ohne Modifikation des Containers nicht NUMA-gerecht umsetzbar.

Um den Wert eines Gitterpunktes im Zielfeld zu berechnen, werden seine vier Nachbarpunkte im Quellfeld benötigt. Diese befinden sich in drei Segmenten, die auf Basis eines Segmentiterators mittels Iteratorarithmetik errechnet werden können. Ein Segmentiterator, der auf das erste Segment der Matrix zeigt, kann aus dem segmentierten Iterator `matrix.begin()` folgendermaßen erzeugt werden:

```
typedef segmented_iterator_traits<seg_array<double>::iterator> traits;
traits::segment_iterator matrixsegiter = traits::segment(matrix.begin());
```

Der segmentierte Iterator liefert neben dem Segmentiterator auch den lokalen Iterator, mit dem die Elemente innerhalb eines Segments adressiert werden können:

```
double* localiterator = traits::begin(matrixsegiter)
```

Die von `segmented_iterator_traits<>` (siehe S. 47) bereitgestellten Funktionen `begin()` und `segment()` tragen ein `static`-Attribut und können deshalb ohne ein Klassenobjekt genutzt werden. Mit diesem Instrumentarium kann die Parallelisierung des Relaxationsverfahrens für den segmentierten Container analog zu der auf Seite 49 vorgestellten Parallelisierung der Vektortriade durchgeführt werden.

4.4 Performancebetrachtung

Die Performancewerte in den Abbildungen 4.2 und 4.3 zeigen, dass alle vier Techniken in etwa gleich gut für die Testanwendung geeignet sind. Bei horizontaler Gebietszerlegung ist die Skalierung bis vier Threads perfekt. Im vertikalen Fall werden durch die Zerteilung der einzelnen Vektoren die Arbeitsschleifen kürzer. Der durch OpenMP bedingte Overhead fällt dadurch stärker ins Gewicht, was eine Performanceverbesserung von nur 80 statt 100 % bei Verdopplung der Prozessorzahl erklärt.
Der Schritt von vier auf acht Prozessoren bringt in beiden Fällen nur geringe Leistungssteigerung, da hier keine weiteren Prozessorsockel und damit Speicherbandbreite hinzukommen, sondern lediglich der zweite Prozessor auf jedem der vier Sockel mit genutzt wird. Abschließend muss betont werden, dass keiner der C++-Container gegenüber dem vermeintlich effizienteren `double`-Array signifikant abfällt.

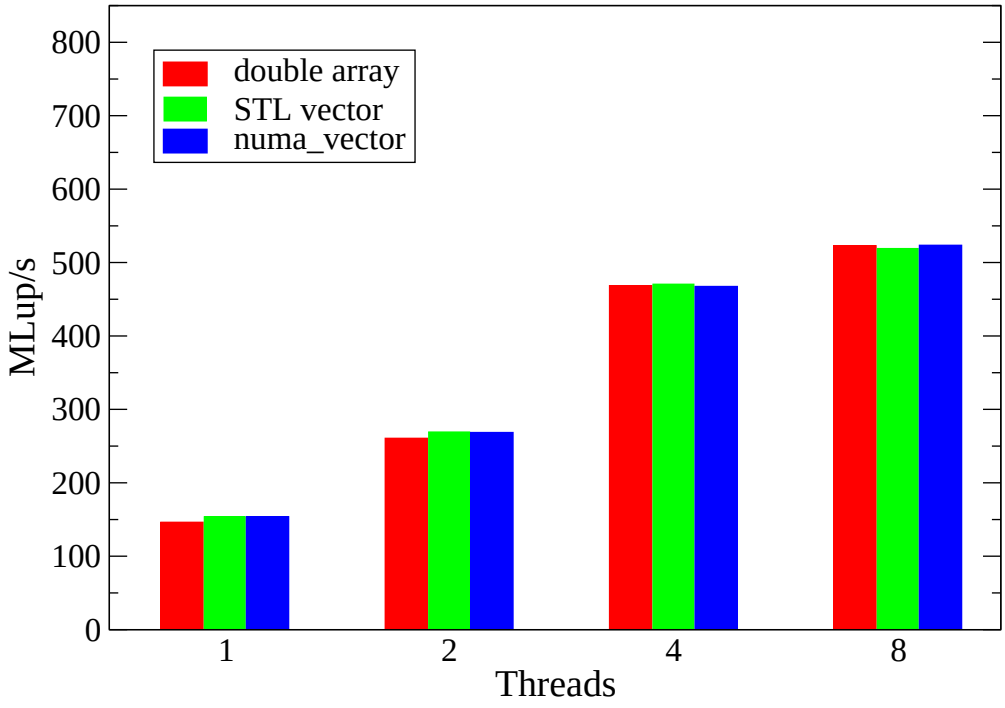

Abbildung 4.2: Performance des Relaxationsverfahrens (Opteron, Gebietszerlegung vertikal, siehe Abbildung 4.1 links)

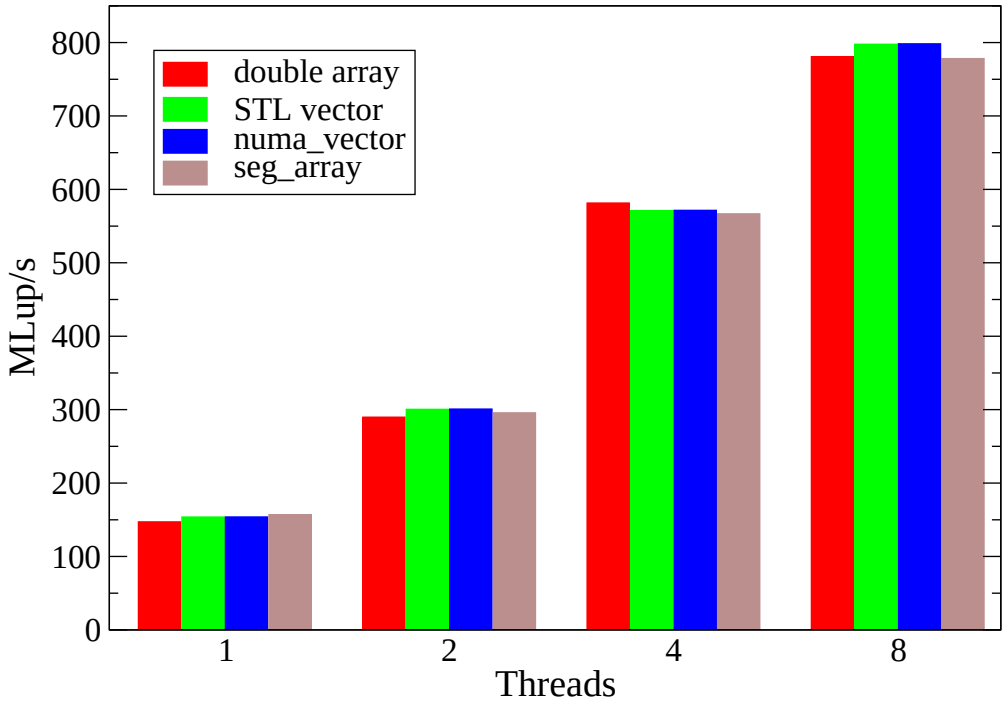

Abbildung 4.3: Performance des Relaxationsverfahrens (Opteron, Gebietszerlegung horizontal, siehe Abbildung 4.1 rechts)

5 Abschluss und Ausblick

5.1 Resümee

Diese Arbeit beschäftigte sich mit Programmiertechniken zur Performanceoptimierung von Shared-Memory-parallelen C++-Programmen auf Systemen mit ccNUMA-Architektur.

Zunächst wurden die beiden prinzipiellen Performanceprobleme auf ccNUMA-Systemen beschrieben und mit Performancedaten belegt: das Lokalitätsproblem und das Bandbreitenproblem. Die prinzipielle Lösung für beide besteht in korrektem Page Placement, dessen Implementierbarkeit von der First-Touch-Strategie bei der Zuordnung virtueller zu physikalischen Speicherseiten abhängt. Diese Bedingung ist bei allen im High Performance Computing eingesetzten Betriebssystemen erfüllt.

Es wurde gezeigt, dass Standard-C++-Programmiertechniken in verschiedenen Ausprägungen, z.B. die Verwendung von Feldern von Objekten oder STL-Containern, korrektes Placement ohne besondere Maßnahmen unmöglich machen. Für einige Szenarien wurden Lösungen erarbeitet, die mit einem Minimum an zusätzlichem Programmieraufwand auskommen und auch in existierenden Codes eingesetzt werden können:

Felder von Objekten, deren Konstruktoren ihre Memberdaten initialisieren, können durch ein Überladen des `new[]`-Operators richtig auf die Lokalitätsdomänen verteilt werden. Das Page Placement findet dann bereits vor der Objektkonstruktion statt.

STL-Container haben die Eigenschaft, mittels eines zusätzlichen Template-Parameters, der Allocator-Klasse, konfigurierbar zu sein, die die Allokation benötigter Speicherbereiche und das Platzieren von Objekten des `value_type` übernimmt. Dadurch kann zumindest für die am häufigsten verwendete `vector<>`-Klasse, deren Nutzdaten ebenso wie bei einem „klassischen“ Feld in einem zusammenhängenden Speicherbereich gehalten werden, ein effizienter Einsatz auf ccNUMA-Systemen ermöglicht werden.

Einige Eigenschaften des `vector<>`-Containers lassen für numerische Anwendungen ein besser angepasstes Werkzeug wünschen, das die Fähigkeiten der `valarray<>`-Klasse hat und zusätzlich durch die Ausstattung mit Iterator-Subklassen mit den STL-Algorithmen kompatibel ist. In Form des `numa_vector<>`-Containers wurde ein solches Werkzeug geschaffen, das neben dem erwünschten NUMA-Placement (konfigurierbar über eine Allocator-Klasse) noch den Nebeneffekt einer effizienter implementierten `operator[]`-Methode besitzt.

Schließlich wurde die von M. Austern [24] entwickelte Idee eines segementierten Iterators aufgegriffen und erstmals auf ihre Leistungsfähigkeit überprüft. Der entwickelte `seg_array<>`-Container ist kompatibel mit STL-Algorithmen, die vorwärts-Iteratoren benötigen, und dank seiner Fähigkeit, Daten mit einem automatischen “Padding” zu speichern, ideal für die

Verwendung auf Systemen mit segmentiertem Speicher geeignet. Dazu gehören nicht nur ccNUMA-Designs, sondern auch in letzter Zeit modern gewordenen Multicore-Prozessoren, deren vielschichtige Cache-Struktur ein besonderes Bewusstsein für Datenlokalität verlangt. Segmentierte Iteratoren erlauben überdies durch den Einsatz von Merkmalsklassen (Traits), Algorithmen zu entwerfen, die sich der Segmentierung bewusst sind und dadurch keine Performanceeinbußen gegenüber der Verwendung mit herkömmlichen Iteratoren zeigen.

In einem Benchmarkvergleich anhand des Gauß-Seidel-Verfahrens konnte abschließend gezeigt werden, dass alle entwickelten Methoden in der Lage sind, die Performance von in reinem C geschriebenem und aufwendig von Hand NUMA-kompatibel gemachtem Code zu erreichen.

5.2 Weiterführende Arbeiten

An viele der hier behandelten Probleme lassen sich weiterführende Projekte anknüpfen. Zu prüfen wäre beispielsweise, inwiefern sich andere STL-Container wie `list<>` oder `deque<>` für die Verwendung mit NUMA-fähigen Allokatoren eignen. Da hier die Daten nicht notwendigerweise in einem zusammenhängenden Speicherbereich abgelegt werden, müssten die `allocate()`- und die `construct()`-Methode Hand in Hand arbeiten, um eine korrekte Zuordnung von Seiten zu Threads zu gewährleisten. Unter der Bedingung, dass ein Allokator keinen internen Status haben darf (alle Allokatoren sind unter `operator==` gleich), ist dies schwer sicher zu stellen. Eventuell wäre sogar die Entwicklung einer eigenen Semantik für Allokatoren und einer passenden Kollektion von NUMA-Containern angebracht.

In der hier behandelten Form weisen der segmentierte Container `seg_array<>` und der zugehörige Iterator `seg_array<>::seg_array_iter` noch einige Schwachstellen auf. Zunächst gibt es nur einen vorwärts-Iterator, der sich zwar einfach auf einen bidirektionalen Iterator verallgemeinern ließe; die Entwicklung eines random-access-Iterators ist jedoch unter genauer Beachtung der Kosten für die Methoden `operator[]`, `operator+=` etc. durchzuführen. Wie in der Arbeit dargelegt, wird man bei der Verwendung segmentierter Iteratoren so weit wie möglich versuchen, sich auf lokale Zeiger zurückzuziehen, um keine Performanceeinbußen hinnehmen zu müssen. Trotzdem ist eine effiziente Implementierung eines wahlfreien `seg_array<>::seg_array_iter` wünschenswert.

Die Benchmarkergebnisse für `seg_array<>` lassen überdies bereits in seriellen Programmen eine starke Abhängigkeit der Ergebnisse — sowohl im als auch außerhalb des Cache — von der gegenseitigen Ausrichtung (Alignment) der Felder erkennen, auf die das Padding ggf. starken Einfluss hat. Eine genauere Untersuchung dieser Effekte auf verschiedenen Prozessor- und Systemarchitekturen wäre angebracht, um die gröbsten Fehler vermeiden zu helfen. Bei parallelen Anwendungen auf ccNUMA-Systemen ist überdies die Größe der Speicherseiten (Pages) von Bedeutung für das Placement. Da im HPC-Bereich die Verwendung großer Pages nicht unüblich ist, wären exakte Daten zu diesem Problemkreis hilfreich.

Alle Placement-Strategien, die hier vorgestellt wurden, haben den Nachteil, dass die Verteilung von Schleifeniterationen auf die Threads, das sogenannte OpenMP-Scheduling, eng auf `static`, ggf. mit einer entsprechenden Blockgröße (`chunk_size`), festgelegt ist. Bei hoch

nichtregulärem Rechenaufwand über alle Iterationen wäre jedoch `dynamic` oder `guided` u.U. besser geeignet. Um unter diesen Bedingungen in gewissen Grenzen trotzdem noch ein gutes Placement sicher zu stellen, könnte man den segmentierten Iterator um eine Komponente erweitern, die für jedes Segment die Zuordnung zu einer Lokalitätsdomäne speichert. Unter Verwendung einer speziellen Methode oder eines Sub-Iterators könnte dann jeder Thread über genau die Segmente Iterieren, die in seiner Lokalitätsdomäne liegen. So ein Programmiermodell wäre zwar etwas weiter von "purem" OpenMP entfernt, könnte jedoch unter den o.A. Bedingungen für eine verbesserte Performance sorgen.

Schließlich ist zu erwarten, dass gutes Placement einen umso positiveren Einfluss auf die Applikationsperformance hat, je langsamer die Verbindung zwischen zwei Lokalitätsdomänen ist. Insbesondere auf Distributed-Shared-Memory-(DSM-)Systemen, wo im schlimmsten Fall eine Softwareschicht dazu dient, ein Cluster als ccNUMA-Architektur zu präsentieren [28], sind Latenzzeiten für nichtlokalen Zugriff in der Größenordnung von vielen tausend Taktzyklen üblich. Speicherseiten, die von zwei Threads "geteilt" werden müssen, führen hier bei lesendem Zugriff zur Replikation, beim Schreibzugriff jedoch zu False Sharing [6], was sich katastrophal auf die Performance auswirkt. Das im segmentierten Container implementierte Padding könnte hier wahrscheinlich in vielen Fällen Abhilfe schaffen. Eine ähnliche Situation ergibt sich bei Multicore-Prozessoren, in denen viele unterschiedliche Cache-Ebenen, manche separat für jeden Core, manche gemeinsam, Daten zwichenspeichern. Auf Ebene der Caches liegt also im Prinzip eine ccNUMA-Architektur vor. In Situationen, wo das komplette Problem in den aggregierten Cache aller Prozessoren passt, sollte False Sharing unbedingt verhindert werden. Dies ist durch entsprechendes Padding ebenfalls zu erreichen.

Literaturverzeichnis

[1] W. E. Nagel, W. Jäger, M. Resch: *High Performance Computing in Science and Engineering '06* (Springer, Heidelberg), 2006.

[2] `http://www.rrze.uni-erlangen.de`

[3] S. Meyers: *Effective C++, 3rd edition* (Addison-Wesley, Massachusetts), 2005.

[4] D. Bulka, D. Mayhew: *Efficient C++* (Addison-Wesley, Massachusetts), 2003.

[5] B. Bergen: *Hierarchical Hybrid Grids: Data Structures and Core Algorithms for Efficient Finite Element Simulations on Supercomputers* (Advances in Simulation, Band AS14), 2005.

[6] G. Hager, G. Wellein: *Concepts of High Performance Computing* (Regionales Rechenzentrum, Erlangen), 2007.

[7] W. Gropp, E. Lusk, A. Skjellum: *Using MPI - Portable Parallel Programming with the Message-Passing Interface* (MIT Press, Massachusetts), 1996.

[8] `http://www.mpi-forum.org`

[9] `http://www.lrz-muenchen.de/services/compute/hlrb/`

[10] W. Schönauer: *Scientific Supercomputing - Architecture and Use of Shared and Distributed Memory Parallel Computers* (self-edition, Karlsruhe), 2000.

[11] J. L. Hennessy, D. A. Patterson: *Computer Architecture. A Quantitative Approach* (Morgan Kaufmann, San Francisco), 2003.

[12] `http://www.top500.org/`

[13] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rüde, G. Hager: *Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method* Proceedings of ICMMES 2006 (accepted for publication in Progress in Computational Fluid Dynamics), 2007.

[14] OpenMP Application Program Interface Version 2.5 May 2005
`http://www.openmp.org/drupal/mp-documents/spec25.pdf`

[15] `http://www.open-mpi.org/software/plpa/`

[16] C. Terboven, D. an Mey: *OpenMP and C++* Proceedings of IWOMP2006 - International Workshop on OpenMP, Reims, France, June 12-15, 2006.
`http://iwomp.univ-reims.fr/cd/papers/TM06.pdf`

[17] S. B. Lippman, J. Lajoie, B. E. Moo: *C++ Primer, 4th edition* (Addison-Wesley, Massachusetts), 2005.

[18] S. Kuhlins, M. Schader: *Die C++ Standardbibliothek, 4. Auflage* (Springer, Berlin), 2005.

[19] `http://incubator.apache.org/stdcxx/doc/stdlibref/allocator.html`

[20] M. H. Austern: *What Are Allocators Good For?* (C/C++ Users Journal), 2000. `http://www.ddj.com/cpp/184403759`

[21] I. Pohl: *C++ for C Programmers, 3rd edition* (Addison-Wesley, Massachusetts), 1999.

[22] `http://www.sgi.com/tech/stl/stl_introduction.html`

[23] `http://www.stlport.org/resources/StepanovUSA.html`

[24] M. H. Austern: *Segmented Iterators and Hierarchical Algorithms* (in M. Jazayeri, R. G. K. Loos, and D. R. Musser (ed.), Generic programming: International Seminar on Generic Programming, Castle Dagstuhl, Springer), 2001.

[25] M. H. Austern: *Algorithms and Containers* (C++ Report 12:7), July/August 2000.

[26] Intel 64 and IA-32 Architectures Optimization Reference Manual
`http://developer.intel.com/design/processor/manuals/248966.pdf`

[27] Software Optimization Guide for AMD64 Processors
`http://www.amd.com/us-en/assets/content_type/...`
`...white_papers_and_tech_docs/25112.PDF`

[28] Cluster OpenMP for Intel Compilers for Linux
`http://www.intel.com/cd/software/products/asmo-na/eng/329023.htm`

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 28. September 2007