TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MAGISTERARBEIT

Perfomanceanalyse von multi-core Prozessoren für Anwendungen aus Computational Science

Ausgeführt am Institut für

Scientific Computing

der Universität Wien
unter der Anleitung von

ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer

durch

Marion Bauer

1100 Wien, Carl-Appel-Str 9/6.3
Mat. Nr.: 0150551
E066 926

Wien, am 12. Mai 2008

Kurzfassung

In dieser Arbeit werden die Performance-Eigenschaften von multi-core Prozessoren für Anwendungen aus Computational Science analysiert. Bei diesen Eigenschaften handelt es sich insbesondere um Ausführungszeiten von Anwendungen, die sehr große Datenmengen in Form von Arrays haben und auf diesen Daten arithmetische Operationen ausführen. Bei diesen Messungen, die auf verschiedene Desktop- und Serverarchitekturen durchgeführt wurden, kann festgestellt werden, wie groß die Bedeutung von guter Datenlokalität in parallelisierten, wissenschaftlichen Anwendungen ist. Weiters ist eine Technik für schlechte Datenlokalität entwickelt worden. Bei dieser Strategie, die darauf abzielt das Verhalten der Caches zu optimieren, wird versucht Prozessor *Stalls* - verursacht durch Cache Misses - zu verstecken indem mehrere Threads auf einen Core zugewiesen werden und somit immer ein bereiter Thread zur weiteren Abarbeitung auf dem Core zur Verfügung steht.

Inhaltsverzeichnis

1. Einleitung

Bis vor wenigen Jahren wurde eine Leistungssteigerung der damals üblichen single-core Prozessoren durch das Erhöhen der Taktfrequenz und durch eine Verbesserung des instruction-level Parallelismus (ILP) durch eine komplexere Hardware erreicht. Bei dieser Art der linearen Weiterentwicklung ist man allerdings einerseits an physikalische Barrieren gestoßen, da eine weitere Steigerung der Taktfrequenzen zu viel Abwärme und auch einen zu großen Stromverbrauch verursachen würde. Andererseits wurden aber auch die Grenzen der Bandweiten auf einem Chip erreicht. Das bedeutet, dass die physikalische Grenze bei der Bandbreite bei der Übertragung zwischen Prozessor und Speicher erreicht wurde.

Die Leistungssteigerung bei dieser Art von Architektur kann daher nicht mehr durch eine Erhöhung der Taktfrequenz und eine bessere Unterstützung von ILP erreicht, sondern nur durch den vermehrten Einsatz von thread-level Parallelismus erreicht werden. TLP ist allerdings deutlich komplexer und kann nicht mehr automatisch durch den Compiler extrahiert werden. So wird eine Mitwirkung des Programmierers nötig um eine Leistungssteigerung zu erreichen. Um allerdings eine Leistungssteigerung durch TLP zu erreichen muss die Applikation bereits bei der Entwicklung parallelisiert werden. Der Programmierer muss also beim Erstellen der Applikation festlegen welche Arbeiten parallel durchgeführt werden können und wo Synchronisationspunkte nötig sind.
Die Komplexität bei der Erstellung von Anwendungen ist bei der Verwendung von paralleler Programmierung, die beim Einsatz von multi-core Architekturen erforderlich ist, ist deutlich größer als bei den bestehenden Softwarenetwicklungs-Paradigmen. Dual-Core Maschinen sind in den letzten Jahren immer üblicher geworden; allerdings ist zu erwarten, dass die Anzahl der Cores mit jeder Prozessorgeneration verdoppelt wird.

Die ersten Kapitel dieser Arbeit beschäftigen sich mit den technischen Grundlagen von Mehrprozessorsystemen, die es schon seit deutlich längerer Zeit gibt als multi-core Architekturen.

Hier wird versucht eine Einordnung der multi-core Architekturen in bereits vorhandene Taxonomien vorzunehmen und dabei ebenso Unterschiede zwischen den Architekturen als auch Gemeinsamkeiten aufzuzeigen.
Bei klassischen Client-Server Architekturen, wie sie vor allem im Bereich der Webapplikationen vorkommen, ist es meist fast ohne zusätzlichen Programmieraufwand möglich die multicore Architekturen sinnvoll zu nutzen, weil jeder Request eines Clients als eigenständiger Thread parallel abgearbeitet werden kann und nur die Schreibzugriffe auf gemeinsame Daten synchronisiert werden müssen. Eine andere Situation entsteht, wenn man Anwendungen aus Computational Science betrachtet. Hier geht es meist darum, dass Berechnungen auf eine große Menge von Daten gemacht werden und es oftmals Abhängigkeiten gibt. Meist ist es so, dass das Ergebnis einer Berechnung den Input für eine weitere Berechnung gibt.

Eine besondere Auswirkung auf die Performance von muli-threaded Applikationen hat die Datenlokalität. In dieser Arbeit wird die Bedeutung von Datenlokalität durch die Durchführung von Experimenten dargestellt. In diesen Experimenten werden die beiden synthetischen Extremfälle von guter und schlechter Datenlokalität nachgestellt. Die Messungen erfolgen dann mit diesen beiden Extremfällen und einigen Zwischenstufen. Diese Experimente werden auf vier verschiedenen multi-core Hardware Plattformen - welche zuvor genau beschrieben wurden - durchgeführt.
Diese Experimente zeigen deutlich, dass ein erheblicher Performancegewinn durch gute Datenlokalität auf multi-core Plattformen erreicht werden kann. Da es aber keine leichte Aufgabe darstellt, das Verhalten der Caches zu beeinflussen und somit Cache Misses zu vermeiden wird in dieser Arbeit nach Strategien gesucht wie man bei schlechter Datenlokalität vorgehen kann. Eine sich bietende Möglichkeit mit schlechter Datenlokalität umzugehen, und die dadurch entstehenden Performanceverluste zu minimieren, ist die auftretenden Cache Misses zu verstecken. Ein guter Ansatz wird hier durch Multithreading geboten. Werden mehrere Threads - die sich eine Aufgabe teilen - auf einen Core zugewiesen und kommt es nun aufgrund eines Cache Misses zu einem *Prozessorstall* sollte, wenn ausreichend Threads auf den Core gelegt wurden, sofort ein anderer, bereiter Thread ausgeführt werden. Unterstützt der Prozessor nun Multithreading, so kann ohne auch nur einen Clock Cycle zu verlieren sofort auf den nächsten, bereiten Thread gewechselt werden, aber auch im Falle von nicht-multithreading Prozessoren dauert eine Migration eines Threads meist kürzer als der *Stall* der durch einen Cache Miss verursacht wurde. Dieses Experiment wurde auch wieder auf allen vier multi-core Architekturen durchgeführt, jeweils für die beiden synthetischen Extremfälle für gute und schlechte Datenlokalität mit jeweils unterschiedlich vielen Threads per Core.

Die Arbeit ist wie folgt aufgebaut:

In Kapitel 2 werden die technischen Grundlagen von High Performance Computing Plattformen erläutert. Dort wird eine sehr allgemeine Taxonomie von Multiprozessorrechnern dargestellt und dann versucht, eine Einordnung der neuen Multi-Core Architekturen in diese Taxonomie vorzunehmen. Des weiteren wird auf das allgemeine Cache Kohärenz Problem in Abschnitt 2.2.3 eingegangen. Da es für die Nachvollziehbarkeit, von in späteren Abschnitten erläuterten Experimenten notwendig ist einen Überblick über das Speicherhierarchiekonzept zu haben, wird dieser kurz in Abschnitt 2.3 umrissen.

In Kapitel 3 wird kurz auf die Bedeutung der Entwicklung von Multi-Core für Scientific Computing Anwendungen eingegangen. Kapitel 4 befasst sich genauer mit den Architekturen der vier in den Experimenten verwendeten Rechnerarchitekturen und ihren Besonderheiten; abschließend wird eine Übersicht über diese Architekturen in Abschnitt 4.5 gegeben.

In Kapitel 5 wird in Abschnitt 5.1 einerseits auf den Aufbau und die Zielführung der beiden Experimente eingegegangen, andererseits wird der Pseudo-Code der Experimente in Abschnitt 5.2 dargestellt. Da es bei der Erstellung der Experimente nötig war einige ungewohnt Funktionen zu verwenden, wird auf den Systemaufruf zum Binden eines Threads an eine bestimmte CPU in Abschnitt 5.2.3 eingegangen. Dieser Systemaufruf kann in einer plattformunabhängigen Sprache wie Java nicht direkt aufgerufen werden, weshalb auch eine Vorstellung des *Java Native Interfaces* - das dazu benötigt wurde - in Abschnitt 5.2.4 aufgenommen wurde.

In Kapitel 6 werden für jede verwende Prozessorarchitektur die Ergebnisse, sowohl als Diagramm als auch als die normalisierten Werte, dargestellt. Die Ergebnisse werden dann in Kapitel 7 näher erläutert und analysiert. Zum Abschluss werden in Kapitel 8, in einer Zusammenfassung, die die wichtigsten Schwerpunkte der Arbeit nochmals herausgestrichen.

2. High Performance Computing Plattformen

2.1. Einführung

Moore's Law (das Gesetz von Gordon Moore, dem Gründer von Intel [17]) besagt, dass sich die Anzahl der Transistoren pro Chip etwa alle 18 Monate verdoppelt[17]. Durch das Erhöhen der Taktfrequenz und der besseren Nutzung von Instruction Level Parallelismus (ILP) erreicht man Leistungssteigerungen bei Prozessoren. Unter ILP versteht man, das parallele Ausführen von Instuktionen (Operationen) gleichzeitig ausgeführt werden können, damit sie einander nicht beeinflussen. Diese Art der Performanceverbesserung wird durch den Compiler extrahiert[7]. Durch diese beiden Arten der Leistungssteigerung sind immer komplexere Prozessoren, die auch einen höheren Stromverbrauch haben, entstanden.

Als eine Taktfrequenz von 4 GHz erreicht wurde, ist die Abwärme und der damit verbundene Energieverbrauch so groß geworden, dass damit nicht mehr sinnvoll umgegangen werden konnte.

Daher muss nach anderen Wegen gesucht werden, um eine weitere Leistungssteigerung zu erreichen. Eine sich bietende Möglichkeit ist die Erhöhung der Anzahl der Prozessorkerne[1] auf einem Chip. Befinden sich mehrere Cores auf einem Chip - wobei hier der einzelne Prozessor meist eine niedrigere Taktfrequenz besitzt als wenn er sich einzeln auf einem Chip befindet - ist der Energieverbrauch deutlich geringer als bei Verwendung eines einzelnen Cores auf einem Chip.
Dadurch entsteht eine neue Generation von Rechnerarchitekuren, die auf Parallelarbeit in Form von Thread Level Parallelismus (TLP) basiert. TLP ist allerdings sehr viel komplexer als

---

[1] Prozessorkern ist die deutsche Übersetzung für *Core*; im weiter folgenden Text wird nur die Bezeichnung *Core* verwendet

ILP und kann üblicherweise nicht mehr durch den Compiler extrahiert werden. Dadurch entstehen erhebliche Auswirkungen auf die Softwareentwicklung, denn nun muss der Entwickler die Regionen, die parallelisierbar sind, in seiner Software definieren und sich um die Synchronisationspunkte kümmern. Es müssen also single-threaded Anwendungen in *multi-threaded* Anwendungen umgeschrieben werden.

Die eigentliche Veränderung in der Architektur seitens der Industrie kam, als 2005 nach IBM und Sun auch Intel begann Rechner herzustellen, die mehrere Cores (Kerne) auf einem *Die*[2] enthalten. IBMs erster multi-core Prozessore war der Power 4; von Sun war es - die auch in Kapitel 4.1 näher vorgestellter - Niagara. Das Ziel dahinter ist allerdings, noch sehr viel mehr Cores auf einem *Die* produzieren zu können[2].

Der Begriff *multi-core* bedeutet, dass sich mehrere Cores auf einem Chip befinden. Zur Zeit befinden sich bis zu maximal 8 Cores auf einem *Die*, wie es bei der Sun Niagara der Fall ist[15]. Die bisherigen Rechner haben immer nur einen Kern pro *Die* gehabt, danach kamen die ersten dual-core Architekturen, die bereits zwei Kerne enthalten; dies kann aber auch noch weiter entwickelt werden - die Sun T1 enthält heute bis zu 8 Cores auf demselben *Die*. Allerdings ist im Moment keine Begrenzung nach oben zu erkennen; mit jedem Schritt in der Weiterentwicklung der Fertigungstechnik kann die Anzahl der Cores auf einem *Die* noch weiter erhöht werden und momentan sieht es nicht so aus als ob hier bald eine Grenze in der Anzahl der Cores pro *Die* zu erwarten ist. Dadurch können in absehbarer Zeit *many-core* Architekturen entstehen[2].

Im Gegensatz zu den bisherigen Leistungssteigerungen - wo eine Leistungssteigerung durch eine Verbesserung des ILPs erreicht wurde und seitens des Programmierers meist nur ein erneutes Kompilieren nötig war - ist bei dieser Art der Rechnerarchitektur eine Leistungssteigerung nur bei Programmen erzielbar, bei deren Entwicklung bereits auf TLP geachtet wurde und der Entwickler bereits bei der Erstellung der Software parallelisierbare Tasks und deren Abhängigkeiten definiert hat. Allerdings werden aber nicht die einzelnen Tasks schneller abgearbeitet - dies geschieht eher langsamer, weil die Taktfrequenz eines einzelnen Cores im Verband eines multi-core Prozessors meist geringer ist als bei einem einzelnen Core - sondern eher die Gesamtaufgabe, weil die voneinander unabhängigen Aufgaben gleichzeitig erfüllt werden können.

Wie aber kann nun ein einzelner Task durch einen multi-core Prozessor schneller werden, wenn die Rechnenleistung eines einzelnen Cores in einem multi-core System geringer ist als

---

[2] *Die* bezeichnet die gemeinsame Platine, auf der sich die Cores befinden. Im physikalischen Sinn handelt es sich hierbei um ein Siliconblättchen.

die eines einzelnen Cores? Die einzige Möglichkeit eine Leistungssteigerung zu erreichen ist nur mehr durch den Entwickler selbst möglich. Es ist nötig bereits bei der Erstellung der Programme zu definieren welche Aufgaben unabhängig voneinander und somit parallel ausgeführt werden können und wo es Abhängigkeiten und damit Synchronisationspunkte geben muss. Der Wechsel von sequenzieller Programmierung zu paralleler Programmierung macht das Entwickeln von einfachsten Applikationen bereits sehr viel aufwendiger. Der Aufwand der durch die Erstellung von paralleler Software entsteht ist meist nicht linear mit der damit verbundenen Leistungssteigerung[3].

Bis vor kurzem war es bei der Entwicklung von normaler Anwendungssoftware - wie Textverarbeitungsprogramme, diverse workflow-unterstützende Softwarelösungen für Unternehmen, u.ä. - nicht von großer Bedeutung, ob diese parallelisierbar sind. Im Bereich der (Web)Server bringt die Verwendung von Mehrprozessormachinen - ebenso wie die Verwendung der Multicores - ohne eine weitreichende Anpassung der Software schon Vorteile; wird jeder Request an den Server als ein eigenständiger Thread behandelt, so kann dieser problemlos parallel zu den anderen Anfragen abgearbeitet werden. Zu synchronisieren sind bei solchen Anwendungen meist nur schreibende Zugriffe auf gemeinsame Datenquellen[4].

Bei komplexeren Anwendungen, wo es nicht um voneinander nahezu unabhängige Anfragen geht sondern um mehrere - meist voneinander abhängige - Berechnungen, ist es nötig seitens des Entwicklers zu definieren, welche der Berechnungen parallelisierbar sind und wo es Synchronisationspunkte geben muss.
Bei den bisher bekannten Mehrprozessorsystemen existieren zwei oder mehr voneinander vollkommen unabhängige CPUs. Diese CPUs sind nur durch ein Interconnection Netzwerk zur Kommunikation verbunden; ansonsten teilen sie sich keine weiteren - auf der CPU lokalisierten - Ressourcen. Bei multi-core Prozessoren ist das ein wenig anders, denn hier befinden sich zwei oder mehr Cores auf einem *Die*. Ein Core ist keine vollständige CPU in dem oben beschriebenen Sinne. Ein Core hat üblicherweise Register und einen privaten L1 Cache, der L2 Cache wird bei den hier verwendeten Prozessoren immer von mindestens zwei Cores gemeinsam genutzt.

Allerdings gibt es im Bereich der wissenschaftlichen Computeranwendungen schon seit langem so rechenintensive Aufgaben, dass diese ohne Parallelisierung nicht mehr in sinnvoller Zeit auf einer einzelnen Ressource zu berechnen sind.

2.2. Rechnerklassifikation

2.2.1. Klassifikation nach Flynn

Die bekannteste und auch einfachste Rechnerklassifikation ist die nach Flynn - sie wurde schon 1966 vorgestellt. In dieser Klassifikation werden die Rechner auf Basis ihrer Befehls- und Datenströme gegliedert. Den Befehlsstrom bilden die Maschinenbefehlsfolgen, die vom Hauptspeicher in das Leitwerk gelangen und von dort als Steuerinformationen an das Rechenwerk weitergeleitet werden. Den Datenstrom bilden die Daten, die als Folgen von Operanden für arithmetische und logische Operationen oder als Folgen von deren Ergebnissen zwischen Hauptspeicher und Rechenwerk übertragen werden.
Die Unterscheidung, die Flynn in seiner Klassifikation trifft, ist das einfache und mehrfache Auftreten der Befehls- und Datenströme womit es zu einer Unterscheidung von vier Klassen kommt[13].

SISD (Single Instruction Stream, Single Data Stream). Diese Klasse umfasst nicht nur den klassischen Universalrechner sondern auch die meisten Einprozessorsysteme, wie sie bis vor kurzer Zeit noch im Bereich der Desktoprechner üblich waren[13].

SIMD (Single Instruction Streams, Multiple Data Stream). Bei dieser Klasse handelt es sich um Parallelrechner, die immer dieselbe Instruction parallel auf unterschiedliche Daten anwenden. Diese Rechner besitzen nur ein Leitwerk - allerdings haben sie mehrere Rechenwerke. Solche Rechner werden Array-Prozessoren oder Feldrechner genannt[13]. Diese Art von Parallelarbeit wird als Daten-Level Parallelismus bezeichnet. Jeder Prozessor hat seinen eigenen Datenspeicher, aber es gibt nur einen - von allen Prozessoren gemeinsam genutzten - Instruktion Speicher. Sind Anwendungen für Daten-Level Parallelismus designed worden, so ist diese Art der Rechnerarchitektur sehr effizent[7].

MISD (Multiple Instruction Streams, Single Data Streams). Diese Klasse ist praktisch von geringerer Bedeutung; bis heute wurde noch kein Rechner mit einer solchen Architektur entwickelt[7]. Sie kann als Makropipelining von MIMD gesehen werden. Ein kontinuierlicher Datenstrom wird nacheinander von Threads bearbeitet und verändert[13].

MIMD (Multiple Instructions Streams, Multiple Data Streams). Jeder der Prozessoren hat seine eigenen Instruktionen, die er auf seine eigenen Daten anwendet. Bei dieser Art der Parallelarbeit handelt es sich um expliziten Thread-Level Parallelismus[7]. In dieser Klasse von Parallelrechnern sind alle Mehrprozessorsysteme einzuordnen. Die Taxonomie lässt allerdings offen, wie die Prozessoren miteinander verbunden sind und wie sie auf den Speicher zugreifen. Somit fallen alle neueren Entwicklungen auf dem Sektor Multiprozessoren in diese Rechnerklasse. Da auch die Art der Verbindung der einzelnen Prozessoren offen gelassen ist, können hier auch Multi-cores eingeordnet werden[13].

Diese Art der Klassifizierung ist ein sehr grundlegendes Model und es gibt Rechnerarchitekturen, die sich nicht klar darin einordnen lassen und deshalb Hybriden aus mehren der oben definierten Klassen darstellen[7].

2.2.2. Multithreading Prozessoren

Die “höhere” Art von Parallelität wird als Thread-Level Parallelität bezeichnet. Dabei wird das Programm in einzelne logische Threads unterteilt, die einen eigenen Prozess darstellen und daher eigene Instruktionen und Daten haben. Hier wird die Verwendung von Parallelarbeit explizit ermöglicht und muss auch im Softwaredesign berücksichtigt werden. Diese Art von Parallelarbeit wird bei typischen Serverapplikationen (viele voneinander fast unabhängige Anfragen werden an einen Server gestellt, es müssen nur die Schreibzugriffe synchronisiert werden) schon durch die Architektur der Software vorgegeben. Bei anderen Applikationen muss sie explizit im Design berücksichtigt werden[7].

Multithreading erlaubt mehreren Threads sich die funktionalen Einheiten eines Prozessors zu teilen. Um das zu ermöglichen muss der Prozessor von jedem Thread eine gesamte Kopie seines Status haben (also eine eigene Kopie des Register Files, des Page Files, ...). Zusätzlich muss auch die Hardware die Möglichkeit bieten zwischen den einzelnen Threads schnell zu wechseln; ein Auswechseln von Threads muss deutlich schneller gehen als das Wechseln eines ganzen Prozesses[7].

Es gibt verschiedene Annäherungsmöglichkeiten an Multithreading. Ein Überblick dazu wird in Abbildung 2.1 gegeben. Hier werden auf der x-Achse die *issue slots* und auf der y-Achse die *Clock Cycles* dargestellt. Jedes der grauen Rechtecke Symbolisiert eine Tätigkeit eines Threads in einem *Issue Slot* während eines *Clock Cycles*. Jeweils ein Rechteck mit demselben Grauton stellt die Zugehörigkeit zu einem bestimmtenn Thread dar.

Einerseits gibt es *coarse-grained Multithreading*(in der Abbildung 2.1 ganz links dargestellt); bei dieser Variante werden die Threads nur ausgewechselt, wenn sie aufgrund eines zeitaufwendigeren Grundes blockieren - wie z.B. ein L2 Cache Miss. Ansonsten bleiben sie in Ausführung bis der *Stall* vorüber ist[7].
Andererseits gibt es *fine-grained Multithreading* - in der Abbildung 2.1 an zweiter Stelle mit dem Untertitel "Fine MT" dargestellt - welches zwischen den Threads nach jeder Instruktion wechseln kann, was zu einer überlappenden Ausführung der Threads führt. Fine-grained Multithreading ermöglicht das Umschalten zwischen den Threads nach jedem Clock Cycle. Einer der Vorteile von *fine-grained Multithreading* ist, dass es die *Stalls* von einzelnen Threads verstecken kann, weil dann die Ausführungzeit für einen anderen Thread genutzt werden kann, ohne einen Clock Cycle für die Migration eines anderen Threads zu verlieren. Der Nachteil dabei ist, dass die Ausführungszeit eines einzelnen Threads verringert wird, weil ein Thread der ohne *Stall* ablaufen könnte wird immer wieder durch die Ausführung der anderen Threads unterbrochen[7].

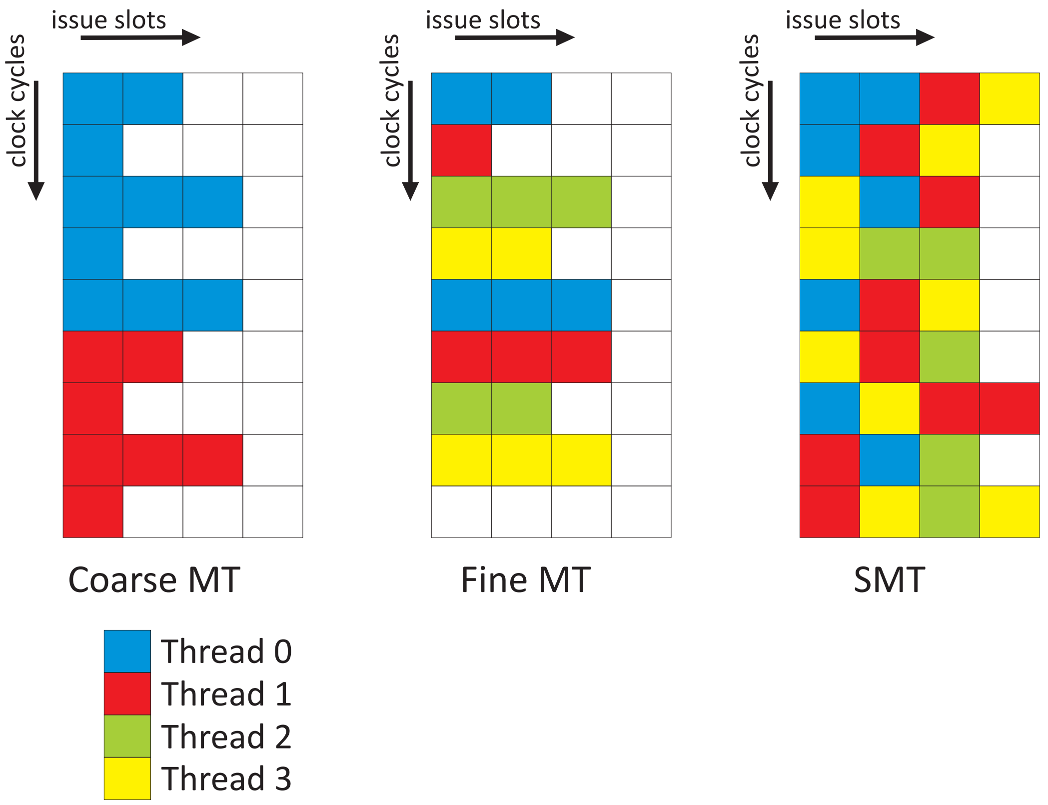


Abbildung 2.1.: Arten von Multithreading[7]

Ganz Rechts wird in der Abbildung 2.1 *Simultaneous Multithreading* (SMT) dargestellt, hierbei handelt es sich um eine Variation von Multithreading, die sich das parallele Vorhandensein von mehreren funktionellen Ressourcen auf einem Prozessor nutzt, die praktisch nie von einem einzelnen Thread gleichzeitig genutzt werden können. Das Ziel vom SMT ist es ILP und TLP gleichzeitig extrahieren zu können. Dies geschieht, indem verschiedenen Threads verschiedene funktionelle Ressourcen innerhalb eines Clock Cycles nutzen[7].

2.2.3. Shared-Memory Multiprozessoren

Als shared-memory Multiprozessoren werden mehrere Prozessoren bezeichnet, die sich einen gemeinsamen logischen oder physikalischen Speicher teilen - der gesamte Adressraum kann über eine einheitliche Adressierung angesprochen werden. Diese Art der Parallelrechner erlaubt eine Skalierbarkeit der Leistung durch das Hinzufügen weitere Prozessoren; allerdings ist diese Art der Skalierbarkeit zwar hoch aber nicht unbegrenzt[13]. Die einzelnen Prozessoren kommunizieren bei dieser Art der Architektur durch die Variablen, die im Speicher abgelegt sind und auf die jeder Prozessor zugreifen kann. Durch dieses gemeinsame Verwenden der Daten muss auch die Art des Zugriffs geregelt sein, weil es sonst zu Inkonsistenzen kommen kann; diese Art der Koordination wird als Synchronisation bezeichnet. Ein solcher Synchronisationsmechanismus wäre z.B. ein *Lock*, während ein Prozessor auf gemeinsame Daten zugreift werden diese gesperrt und der nächste Prozessor kann erst darauf zugreifen, wenn der erste fertig ist und die Sperrung wieder aufgelöst hat.[7].

Grundsätzlich können zwei verschiedene Typen von *shared-memory Multiprozessoren* unterschieden werden:

- **UMA oder SMP**
  UMA steht für *Uniform Memory Access* und SMP steht für *Symmetric Multiprozessor*. Wie bereits der Name besagt, ist die Zugriffszeit hier von jedem Prozessor gleich groß, auch wenn der Speicher in verschiedene *Memory Banks* gegliedert ist[6].

- **NUMA**
  NUMA steht für *Non-Uniform Memory Access*. Hier ist die Zugriffszeit auf bestimmte Speicherbereiche abhängig von dem Prozessor, von dem aus darauf zugegriffen wird. Meistens ist dieser Speicher physikalisch verteilt und es wird auf nicht prozessoreigenen Speicher über ein Verbindungsnetzwerk zugegriffen. Durch diese Art des Speicherzugriffs sind größere Anforderungen an den Entwickler gestellt, allerdings können bei NUMA Architekturen mehr Prozessoren zusammengeschlossen werden als bei UMA Architekturen; dadurch ist theoretisch auch ein größerer Performancegewinn möglich[6]. Der Vorteil bei dieser Architektur ist, dass der Zugriff auf lokale Speicherbereiche eines Prozessors deutlich schneller erfolgt als bei UMA Architekturen[7].

Cache Kohärenz

Durch das Ablegen von gemeinsamen Daten in den lokalen Caches der einzelnen Prozessoren ergibt sich ein neues Problem, weil die Werte von einzelnen Daten auf den lokalen Caches der verschiedenen Prozessoren unterschiedlich sein kann. Werden nun keine weiteren Vorkehrungen getroffen, so hat am Ende jeder der Prozessoren, die diese Daten in ihren lokalen Caches haben, einen anderen Wert. Dieses Problem wird grundsätzlich als *Cache Kohärenz Problem* bezeichnet[7].

Es kann davon ausgegangen werden, dass ein Datensystem kohärent ist, wann immer der letzte geschriebene Wert einer Variable von allen Prozessoren bei einem Zugriff darauf zurückgeliefert wird. Diese relativ einfache Definition ist in der Realität deutlich komplexer. Als *Kohärenz* wird der Wert, der für eine Variable zurückgegeben wird, bezeichnet. Es geht also darum, dass nach einem schreibenden Zugriff alle lesenden Zugriffe von dem Update der Variablen erfahren haben und nun den neuen Wert zurückgeben[7].

In kohärenten Multiprozessorarchitekturen kann es mehrere Kopien von den selben Variablen in verschiedenen Caches. Daher müssen die Caches *Migration* und *Replikation* von gemeinsamen Daten unterstützen.
Migration bedeutet, dass eine Variable transparent zum lokalen Speicherbereich übernommen und von dort mit kleiner Latenzzeit gelesen werden kann. Dadurch wird sowohl die Zugriffszeit auf diese Daten verringert als auch die Belastung der Bandbreite zum gemeinsamen Speicher reduziert.
Replikation bedeutet, dass einzelne Daten, wenn sie vom mehreren Prozessoren gleichzeitig benötigt werden, in verschiedenen Caches doppelt abgelegt werden können[7].

Um nun Inkonsistenten zu vermeiden, wird ein *Cache Kohärenz Protokoll* benötigt. Das Wichtigste bei der Implementierung eines solches Protokolles ist, dass immer der Status von jedem gemeinsamen Datenblock bekannt sein muss[7].

Es gibt zwei verschiedene Arten von Protokollen um das zu erreichen:

- **Directory Based**
  Der Status jedes Datenblocks ist ein einem zentralen Verzeichnis abgelegt. Diese Art hat einen größeren Overhead als der mittels Snooping implementierte Typ, allerdings ist es leichter skalierbar[7].

- **Snooping**
  Jeder Cache, der eine Kopie eines Datenblocks aus dem gemeinsamen Speicher hat, hat auch eine Kopie des Statusblockes dieses Datenblocks. Es gibt aber keinen weiteren zentralen Status zu diesem Datenblock. Die Caches sind alle über ein broadcast Medium erreichbar und alle Cache Controller monitoren auf diesem Broadcast, ob ein Block, dessen Kopie sie haben, angefordert wird, weil er sich verändert hat[7].

2.2.4. Message-Passing Multiprozessoren

Diese Art der Multiprozessoren hat für diese Arbeit nur eine untergeordnete Bedeutung und ist auch nur der Vollständigkeit halber hier erwähnt. Message-passing Multiprozessoren haben keine gemeinsamen Speicherbereiche. Jeder Knoten eines solchen Systems hat einen eigenen Speicherbereich, in dem die zur Bearbeitung verwendeten Daten abgelegt werden. Diese Knoten sind über ähnliche Verbindungsnetzwerke verbunden wie die oben beschriebenen NUMA Architekturen und tauschen gemeinsame Daten über das Kommunikationsnetzwerk aus. Diese nachrichtenbasierte Kommunikation muss explizit durch “message-passing” programmiert werden. Die Hardware und das Betriebssystem stellen einfach Methoden für das Versenden und Empfangen von Nachrichten zur Verfügung[13].

2.2.5. Klassifizierung von Multi-cores

Können multi-core Architekturen auch als moderne SMPs gesehen werden? Einerseits sind es mehrere gleiche Cores, die auf den gleichen physischen Speicherbereich zugreifen. Andererseits gibt es auch bedeutende Unterschiede zwischen Multi-cores und traditionellen SMPs. Multi-cores haben Cores auf einem gemeinsamen *Die* zusammen mit dem Cache. Somit tritt also das Cache Kohärenz Problem immer nur zwischen Cores auf demselben *Die* auf. Dadurch kann es zu deutlich effizienteren Cache Protokollen kommen als in traditionellen SMPs mit längeren Latenzzeiten zwischen den einzelnen Prozessoren. Des Weiteren teilen sich Multicores Hardware Ressourcen wie den L2 Cache; dies ist bei traditionellen SMPs nicht der Fall. Mit der Zeit scheint die Anzahl der Cores auf demselben *Die* sich immer mehr der Anzahl der Prozessoren in einem SMP anzunähern. Momentan sind nur wenige Cores auf einem *Die* zu finden, allerdings sieht es so aus, als würde es in der Zukunft möglich sein, Dutzende von Cores zu produzieren[5].

2.3. Speicherhierarchien

Von der frühen Zeit der Computer bis heute wollen Softwareentwickler unbegrenzte Mengen von schnellen Speichern. So wurde der Wunsch nach einem Speicherkonzept, welches die Vorteile von Datenlokalität und den Trade off zwischen Kosten und schnellem Speicher ermöglicht, laut.
Das *Prinzip der Datenlokalität*([7] Chapter 1) besagt, dass die meisten Programme Daten und Befehle die sie verwendet haben auch wieder verwenden werden. Es können zwei verschiedene Arten von Datenlokalität beobachtet werden. Einerseits gibt es *temporal locality*, dies bedeutet, dass auf Objekte auf die vor kurzer Zeit zugegriffen wurde vermutlich bald wieder zugegriffen wird; hier geht es also um den zeitlichen Aspekt. Andererseits kann *spatial locality* beobachtet werden, darunter wird verstanden, dass Objekte die in benachbarten Speicherbereichen liegen eher aufeinander referenzieren als Objekte die das nicht tun; hier geht es um den Aspekt der physikalischen Anordnung von Daten. Dieses Prinzip und der Grundsatz, dass schnelle Speicher teuerer sind, führt zu einem hierarchischen Speichermodell. Jeder diese Ebenen (Levels) wird kleiner und schneller und somit auch mit jedem Byte teurer als der Vorhergehende[7].
Die einzelnen Levels sind meist so aufgebaut, dass sich alle Daten, die im darüber liegenden, kleineren Speicher liegen, sich auch immer im darunter liegenden, größeren Speicher befinden. Dies wird bis zum Größten, langsamsten Speicher durchgezogen; in diesem befinden sich dann alle Daten, die in dieser Hierarchie vorhanden sein sollen[7].

Die Bedeutung der Speicherhierarchien hat sich mit der zunehmenden Prozessorgeschwindigkeit immer mehr gesteigert, weil der Unterschied zwischen den Speicherzugriffszeiten und den Prozessorgeschwindigkeiten immer größer wurde. So musste durch Architekturansätze versucht werden diesen Unterschied zu überbrücken[7].

Ein Cache ist ein schneller Zwischenspeicher innerhalb der CPU oder in CPU Nähe, der sowohl Befehle als auch Operanden von Programmen bereit halten soll. Jeder Cache besteht aus einem Datenbereich und einem Identifikationsbereich. Die Datenmenge, die ein solcher Cache halten kann, ist meist deutlich geringer als die Datenmenge, die der Hauptspeicher halten kann. "Für jeden Cache sind strukturelle Parameter, eine Organisationsform und eine Ersetzungsstrategie festzulegen."[13]

Heute es es üblich, dass Computersysteme mehrere Cache-Ebenen besitzen. Neben den Registern, die auf der CPU selbst enthalten sind, gibt es in der ersten - somit der CPU am nächsten - Hierarchiebene den **L1-Instruktion-Cache**. Dieser Cache ist von der Speichergröße her am kleinsten und hat dabei die geringste Latenzzeit. Der Zwischenspeicher befindet sich meistens im Prozessorkern und erlaubt einen sehr schnellen Zugriff auf wiederholt auftretende Befehle[13].
Der **L1-Datencache** ist ebenfalls ein Zwischenspeicher, der sich im Prozessorkern selbst befindet. Dieser ist dafür gedacht einen sehr schnellen Zugriff auf immer wieder benötigte Operanden zu gewährleisten[13]. Bei vielen Prozessorbeschreibungen allerdings wird kein Unterschied mehr gemacht zwischen dem Instruktion- und dem Daten-Cache; hier wird dann meistens nur von einem L1-Cache gesprochen.

Bei erfolglosen Zugriffen auf den L1-Cache, weil dieser nur eine begrenzte Kapazität hat, kommt es zu *Cache Misses*. Dies bedeutet, dass die Daten aus dem bedeutend langsameren **L2-Cache** geholt werden müssen. Der L2-Cache ist größer als der L1-Cache, allerdings ist die Zeit, um die Daten aus dem Cache zu holen, hier bereits deutlich länger.
In den neueren Architekturen kann es nach dem L2-Cache auch noch ein L3-Cache - dieser ist wiederum länger als der L2-Cache und die Zeit um Daten von dort abzufragen ist abermals größer - geben, wie es beim Power 5+ der Fall ist. Diese befinden sich allerdings nicht immer gemeinsam mit der CPU auf dem Chip.
Nach dem L2-Cache oder in machen Fällen eben dem L3-Cache, wird als nächstes auf den deutlich langsameren Hauptspeicher zugegriffen. Sollten sich die gesuchten Daten auch nicht im Hauptspeicher befinden, so ist ein Plattenzugriff erforderlich[13].

Ein Cache Eintrag (Cache Line oder Cache Block) ist eine feste Zahl von n aufeinander folgenden Daten- oder Befehlsbytes. Die Anzahl n wird als Block Size oder Line Size bezeichnet. Bei einer Cache Aktualisierung werden jeweils n Bytes aus der nächsten, tiefer gelegenen Speicherhierarchieebene in einen Cache Eintrag übertragen[13].

Bei einem erfolgreichen Lesezugriff auf einen Cache spricht man von einem **Read Hit**. Dieser wird in das Datenregister oder in den Befehlspuffer der CPU geladen und von dort aus weiter verwendet. Gibt es keine Übereinstimmung mit der Identifikation und den sich im Cache befindlichen Daten, so kommt es zu einem **Read Miss**; die Daten oder Befehle müssen aus einer tiefer gelegenen Ebene der Speicherhierachie geholt werden[13].

Als **Write Hit** wird ein erfolgreicher Schreibzugriff bezeichnet. Dabei werden Daten erfolgreich im Cache gefunden und direkt dort verändert. Dabei muss allerdings sichergestellt werden, dass die Cache Bereiche, die schreibend verändert werden, auch wieder in den Hauptspeicher zurückgeschrieben werden[13].

Für das Zurückschreiben in den Hauptspeicher gibt es zwei verschiedene Strategien:

- **Write Back**
  Bei dieser Strategie werden - auch nach einem Schreibzugriff - nur die Daten im Cache verändert; der Datenblock im Hauptspeicher behält seinen ursprünglichen Wert und wird erst dann geschrieben, wenn der veränderte Block aus dem Cache verdrängt wird[7].
- **Write Through**
  Dies ist die einfacher zu implementierende Strategie, hier werden immer alle veränderten Daten in den Hauptspeicher zurückgeschrieben[7].

Gibt es einen erfolglosen Schreibzugriff, so spricht man von einem **Write Miss**. Hier gibt es zwei mögliche Implementierungstechniken:

- *Write Allocate*
  Der Eintrag, der verändert werden soll, wird aus den tieferen Ebenen der Speicherhierachie geholt und von dort in den Cache geschrieben und dort - wie bei einem Write Hit - verändert.
- *Write Around*
  Da der zu verändernde Eintrag nicht direkt von der CPU benötigt wird, reicht es, ihn nur in der tiefer gelegenen Ebene der Speicherhierarchie zu verändern[7].

3. Aspekte für Scientific Computing Anwendungen

Die Entwicklung von multi-core Architekturen hat erhebliche Auswirkungen auf die gesamte Softwareentwicklung. Durch diese Entwicklung kann es zu einer Unterbrechung der Kontinuität in der Entwicklung von High Performance Computing (HPC) Anwendungen kommen, weil in den letzten Jahrzehnten mit jeder Weiterentwicklung der Prozessoren gleichzeitig eine Performancesteigerung mit keinen oder eher geringen Anpassungen der bestehenden Software verbunden war. Durch das Erscheinen von Multi-cores hat sich diese Situation geändert, weil es nun verschiedenartigere Prozessordesigns gibt.

Aspekte die Multi-cores mit sich bringen:

- Geringere Taktfrequenz gegenüber herkömmlichen Single Cores
- Flaschenhals zwischen Hauptspeicher und Prozessoren.

Durch diese beiden Aspekte wird nach guter Datenlokalität - um möglichst wenige Cache Misses zu haben und somit einen effizienteren Speicherzugriff - und somit nach effizienterer Parallelisierung verlangt. Auch effizientere Speicherzugriffe, um Performanceverluste durch Speicherzugriffe so gering wie möglich zu halten, sind erforderlich.

Wie bereits in Kapitel 2 erläutert, haben multi-core Architekturen in manchen Aspekten vergleichbare Eigenschaften wie SMPs. Bei traditionellen SMP Maschinen ist Performance besonders in Verbindung mit Datenlokalität und Load Balancing zu sehen. Somit ist es von großer Bedeutung, dass jeder Core eine vergleichbare Menge an Arbeit zu verrichten hat.

Im folgenden Kapitel wird primär auf den Aspekt der Datenlokalität eingegangen. Bei den traditionellen SMP - mit physisch verteiltem Speicher (NUMA Architekturen) - wird gute Datenlokalität vorallem dadurch erreicht, dass Daten so auf den Speicher verteilt werden, dass es zu möglichst wenigen Zugriffen mit einer langen Latenzzeit kommt.

Die Multi-Cores, die für diese Experimente verwendet wurden, haben allerdings eine zentralisierte Speicherstruktur. Daher werden alle Datentransfers implizit durch Laden und Speichern durchgeführt, wobei es theoretisch jedes mal zu einem Cache Miss kommen kann und damit zu einem Laden von Cache Blöcken. Somit spielt die physikalische Verteilung der Daten keine Rolle bei der Performance von multi-core Architekturen. Die *Spatial Locality* hingegen ist auf verschiedene Art und Weise von Bedeutung.

Um eine Gute Performance auf multi-core Architekturen zu erreichen ist es nötig, dass alle Cores ausgelastet sind und somit ist ein gutes Load Balancing zwischen den einzelnen Cores erforderlich. Des weiteren ist es von Bedeutung, dass bereits bei der Implementierung einer Applikation versucht wird eine gute Datenlokalität zu schaffen um nun Cache Misses zu minimieren.

Für Multi-cores ist die Organisation der Cache Ebenen und ein effizientes Cache Kohärenz Protokoll von größter Bedeutung. Jeder Cache Miss resultiert in einem blockierenden Thread; diese führen zu einer schlechteren Performance.
Allerdings ist es für den Programmierer selbst nicht sehr einfach das Verhalten der Caches zu beeinflussen. Als Beispiel wäre der Sun UltraSPARC T1 (näher beschrieben in Abschnitt 4.1), wo sich alle Cores mehrere L2 Cache Banks teilen, angeführt. Die Zuordnung auf die einzelnen L2 Cache Banks wird von der Hardware vorgenommen und es ist nicht möglich einzelne Cache Banks bestimmten Cores zuzuordnen, um den Traffic auf dem Interconnection-Netzwerk zu reduzieren. Des weiteren können Daten, die auf verschiedenen Cores benötigt werden und sich durch Zufall auf demselben Cache Block befinden zu *false sharing* führen. Bei *false sharing* werden Daten, die sich auf denselben Cache Blöcken befinden, benötigt. Jedes mal wenn es zu einem Schreibzugriff durch einen anderen Prozessor kommt, wird der gesamte Cache Block als ungültig markiert. Somit kommt es beim nächsten Zugriff wieder zu einem Cache Miss, obwohl der Wert, der eigentlich benötigt wird, nicht verändert worden ist.

Da das Verhalten der Caches sehr komplex ist, soll zuerst einmal die Bedeutung der Datenlokalität für Scientifc Anwendungen dargestellt werden. Das Ziel ist herauszufinden, wie viel Performancegewinn durch gute Datenlokalität im besten Fall erreicht werden kann.
Als nächstes wird versucht eine Möglichkeit zu finden, wie man mit schlechter Datenlokalität umgehen kann, weil diese nicht immer vermeidbar ist.
Grundsätzlich können Performanceverluste vermieden werden, indem die Rechenzeit, die ungenützt bleibt wenn ein Thread nach einem Cache Miss *stalled*, durch einen anderen bereiten Thread genutzt wird. Bei dieser Methode ist es besonders gut, wenn der verwendete Multi-core

Multithreading Support anbietet. Unter *Multithreading* Support ist zu verstehen, dass der Core von mehreren Hardware Threads geteilt wird; es kann also zu jedem Taktzyklus ein anderer Thread abgearbeitet werden, ohne dass dadurch Migrationsaufwand entsteht. Einen solchen *Multithreading* Support bietet der Sun T1, der pro Core 4 Threads unterstützt. Dadurch hat man bei 4 Threads auf 8 Cores 32 logische CPUs vom Betriebssystem erkannt, wobei CPU 0 bis 3 auf dem ersten Core laufen, CPU 4 bis 7 auf dem zweiten Cores und so weiter[15].

Zusammenfassend wäre anzuführen, dass in dieser Arbeit zwei Möglichkeiten erläutert werden, wie mit Cache Misses und den daraus resultierenden Performanceverlusten umgegangen werden kann:

1. Vermeidung von Cache Misses
   Ein sicherlich offensichtlicher Ansatz ist Cache Misses zu vermeiden und damit eine klare Aufteilung der Arbeit für die einzelnen Threads zu finden. Das bedeutet, dass jedem Prozessor ein eigenes Set an Daten für eine einzelne Berechnungsphase zur Verfügung gestellt wird. Der gemeinsame Lesezugriff ist normalerweise kein Problem; problematisch sind nur Schreibzugriffe auf gemeinsame Daten. Denn Schreibzugriffe auf Daten, die von mehreren Prozessoren geteilt werden, resultieren meist in Cache Misses und dem nachfolgenden Traffic auf dem Interconnection-Netzwerk. Ebenso kommt es immer wieder zu Sperren

   a) während ein anderer Thread schreibend auf die Daten zugreift und

   b) der nachfolgenden Synchronisation.

   Ein besonders Problem stellt hier *False Sharing* dar; dieses kann nicht nur durch die Anwendung selbst entstehen sondern ist auch abhängig von der Cache Line Size. Es wurden bereits verschiedene Techniken entwickelt um Cache Misses zu vermeiden bzw. zu reduzieren.

2. Verstecken der Cache Misses
   Da es für einen Programmierer keine einfach Aufgabe ist das Verhalten der Caches zu beeinflußen und Cache Misses zu vermeiden, wird versucht, den Performanceverlust durch Cache Misses zu reduzieren, indem man versucht, die Cache Misses zu verstecken. Ein einfacher Versuch basiert dabei auf *Multithreading*. Kommt es zu einem *Stall*, so wird einfach mit einem anderen bereiten Thread weitergearbeitet. Diese Variante ist allerdings nur erfolgreich, wenn darauf geachtet wird, dass mehrere Threads auf einen Core zugewiesen werden und dabei die Ressourcen dieses Cores sinnvoll genützt

werden. Diese Variante scheint besonders für den Sun UltraSPARC T1 (näher beschrieben in Abschnitt 4.1) und dem IBM Power 5+ (näher beschrieben in Abschnitt 4.3) empfehlenswert.

4. Multi-core Prozessoren in den Experimenten

Die Experimente wurden auf vier sehr unterschiedlichen multi-core Prozessoren durchgeführt. Zwei davon waren Server Multi-cores von Sun und IBM und zwei Desktop Multi-cores von AMD und Intel:

- Sun UltraSpark T1 Niagara
- IBM Power 5+
- AMD Opteron Dual Core und
- Intel Core 2 Extreme

In diesem Kapitel werden die Prozessoren im Detail beschrieben, und zum Schluss in einer Tabelle die Leistungsmerkmale der einzelnen Prozessoren zusammengefasst.

4.1. Sun UltraSPARC T1 Niagara

Der Sun T1 bietet nur sehr schlechte Unterstützung für Instruction Level Parallelismus (ILP). Er bietet aber eine ausgezeichnete Unterstützung für Thread Level Parallelismus. Der T1 von Sun ist mit vier, sechs oder acht Cores verfügbar, wobei jeder Core eine Hardwareunterstützung für vier Threads bietet und sich alle Cores auf demselben *Die* befinden. Für die Experimente wurde ein Sun T1 mit 6 Cores - aufgesetzt mit Sun Solaris 10 - verwendet. Auf dieser Maschine wurden die Experimente in Java und in C durchgeführt. Für die Kompilation von Java wurde das Java Development Kit von Sun in der Version 1.5.0; für die Kompilation von C die Sun SPARCworks Entwicklungsumgebung verwendet.

Der Sun T1 unterstützt Multithreading und bietet Unterstützung für bis zu vier aktiven Threads pro Core, die sich eine Pipeline teilen. Diese in der Hardware verankerten Threads werden

von Sun als “Hardware Strands” oder auch einfach nur “Strands” bezeichnet. Jeder dieser vier Hardware Strands befindet sich im Scheduling und wird nach dem Round Robin Prinzip abgearbeitet.
Blockiert ein Hardware Strand, weil er auf eine Ressource wartet, so wird er aus der Pipeline herausgenommen und der Platz an einen anderen bereiten Thread vergeben. Ist der blockierende Thread wieder bereit, so wird er wieder in die Pipline aufgenommen. Die Strands können bei jedem Clock gewechselt werden, ohne dass dafür auch nur ein weiterer Clock für die Migration des Strands erforderlich wird.
Die Hardware Strands werden von Solaris als eigene CPUs erkannt. Wird nun auf einer Sun Niagara mit sechs Cores ein *mpstat* oder ein *psradm* ausgeführt, so werden 24 CPUs angezeigt.

Die einzelnen Cores sind durch eine high-speed Crossbar mit einer geringen Latenzzeit aus Silicon verbunden. Daher handelt es sich bei der UltraSparc T1 kann um einen SMP auf einem Chip[15].

Wie in Abbildung 4.1 dargestellt, besteht jeder Core aus einen Instruction Cache und einen Data Cache, die von den vier Hardware Strands geteilt werden. Alle Cores teilen sich einen gemeinsamen L2 Datencache, der sich direkt auf dem Chip befindet. Der Zugriff auf diesen Datencache erfolgt nach dem UMA (Unifom Memory Access) Prinzip, wobei die Latenzzeit für Zugriffe auf den L2 Datencache von jedem Core gleich lange ist.
Für alle Cores gibt es nur eine Floating-Point Unit, die den Sun UltraSpark T1 eher ungeeignet für High Performance Anwendungen erscheinen lässt, weil es sich bei solchen Anwendungen meist um Floating-Point-intensivere Berechnungen handelt als bei normalen Serveranwendungen. Bei dem nachfolgenden Prozessormodell der UltraSparc T1, der Sun UltraSpark T2, soll aber bereits jeder Core über eine eigene Floating Point Unit verfügen[12].

4.2. AMD Opteron

Der Opteron, der auch den Codenamen “Hammer” trägt, kam in der ersten Jahreshälfte 2003 als erster 64-Bit Prozessor von AMD auf dem Markt. Der Opteron Server Prozessor wurde als erste CPU mit x86-64-ISA im Jahre 2000 vorgestellt[1]. Ebenso gab es auch einige Monate später eine Desktop Variante von diesem Prozessor, den Athlon 64[13].
Im Gegensatz zu Intel, die mit der IA-64-ISA gleich einen kompletten Schritt in die 64 Bit Welt wagten, hat AMD versucht, einen langsameren Übergang mit der Binärkompatibilität,

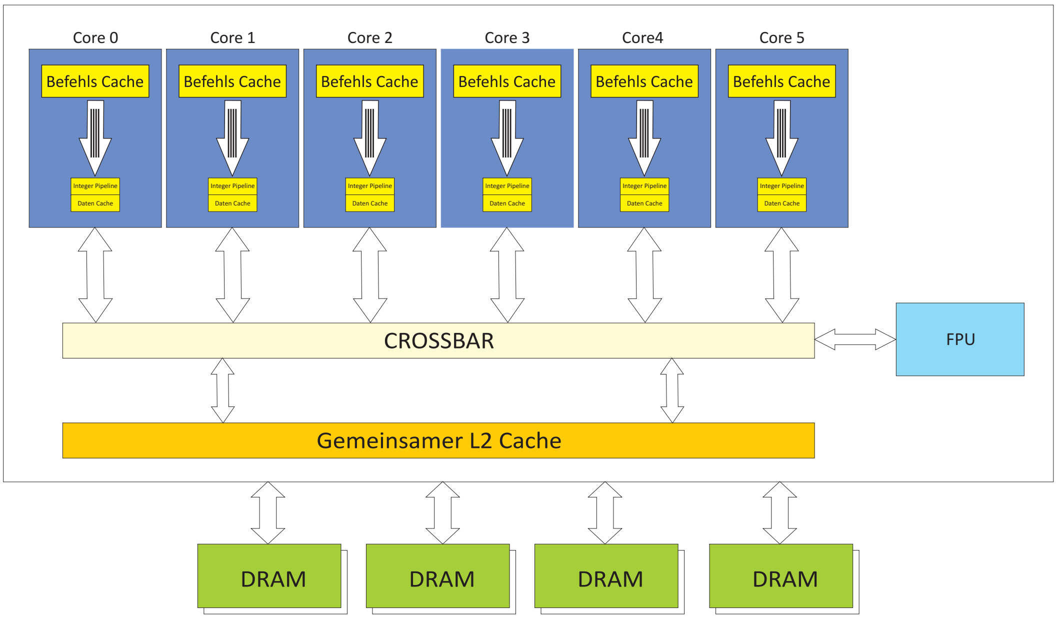


Abbildung 4.1.: Blockdiagramm Sun UltraSparc T1 [15]

die zwischen den alten 32 Bit Architekturen und der neueren 64 Bit Architektur existiert, zu schaffen.
Der verwendete Opteron war Teil eines HPC Systems; dieses besteht aus jeweils zwei Nodes mit jeweils 4 Opteron Kernen - für die Experimente wurde ein einzelner Opteron von einem solchen Knoten verwendet[1]. Auf dem Opteron wurden die Experimente mit dem CC Kompiler von Sun SPARCworks kompiliert; als Betriebssystem lief Sun Solaris.

Im Mai 2005 wurde der Opteron erstmals als Multi-core produziert. Beim Opteron bedeutet der Begriff Multi-core eigentlich immer Dualcore, weil er zwei Cores besitzt.

Auf den ersten Blick unterscheidet sich der Hammer Core nur wenig von seinem Vorgänger, dem Athlon-Prozessor. Allerdings hat er im Vergleich zum Athlon-Prozessor einen deutlichen Leistungsschub, wobei die Ursachen dafür in der Microarchitektur des Hammer Prozessors liegen.
Wie in Abbildung 4.2 dargestellt befindet sich auf dem Chip neben jeweils 64 KB großen L1 Daten- und Befehlscaches auch ein 1 MB großer Write-Back L2 Cache pro Core (integriert). Die Latenzzeit für den Zugriff auf den L1 Datencache beträgt 2 Clocks; bei dem L2 Datencache beträgt sie 7 Clocks. Die Cache Line Größe für den L1 und L2 Cache beträgt 64 bytes. Der AMD Opteron unterstützt kein Multithreading und ermöglicht daher auch nur einen Thread

---

[1] für nähere Informationen siehe http://luna.cs.univie.ac.at

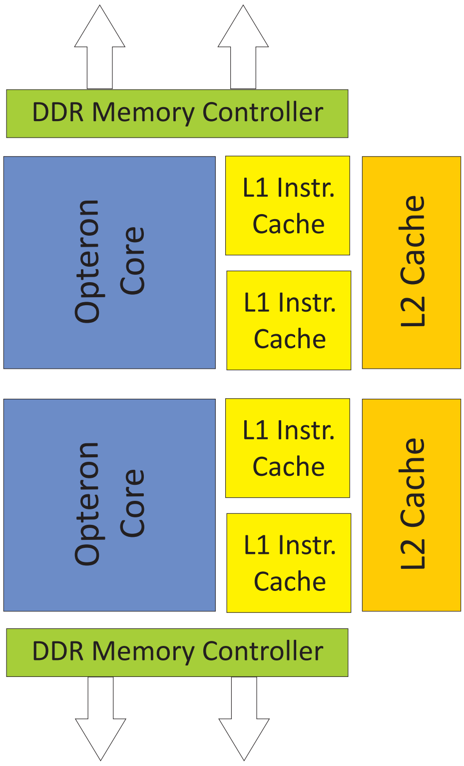


Abbildung 4.2.: Blockdiagramm AMD Opteron [1]

pro Core. Drei Instructions per Clock werden abgearbeitet[1].

4.3. IBM Power 5+

Die IBM Power 5+ besteht aus 2 Dual Core Chipsätzen in einem Package. Es sind 4 Cores enthalten, wobei sich jeweils zwei auf demselben *Die* befinden. Jeder dieser Cores unterstützt Multithreading mit jeweils zwei Threads, die von IBM auch manchmal als “Instruction Streams” bezeichnet werden; pro Core erkennt daher das Betriebssystem den Power 5+ mit 8 CPUs.
Jeweils zwei Cores befinden sich auf demselben *Die* und jeder dieser Cores hat einen eigenen L1 Cache, einen gemeinsamen L2 Cache und einen L3 Cache, der sich nicht mehr auf dem Chip befindet.
Die Power 5+ war für die Durchführung der Experimente mit dem IBM eigenen Betriebssystem AIX aufgesetzt. Die Experimente wurden mit dem GNU gcc Kompiler kompiliert.

Die Power 5+ bietet volle Binär- und Strukturkompatibilität mit Power 4, was bedeutet, dass

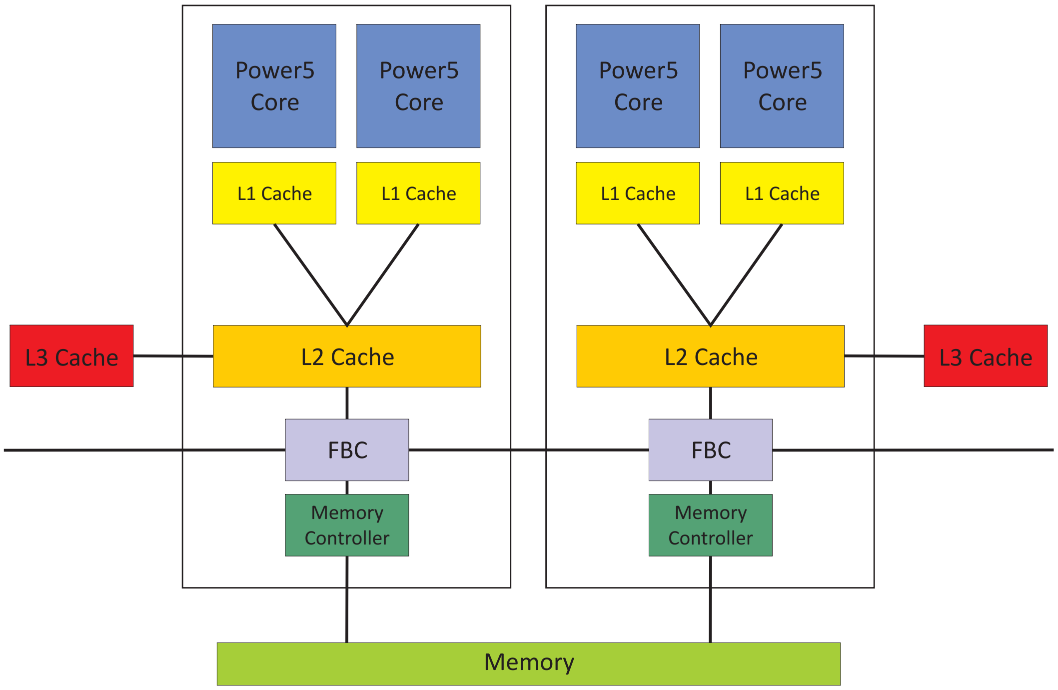


Abbildung 4.3.: Blockdiagramm IBM Power 5+ [16]

nicht nur alle Applikationen des Power 4 ausgeführt werden können, sondern auch dass die für den Power 4 durchgeführten Optimierungen immer noch sinnvoll sind.
Durch diese Basisanforderungen ergaben sich folgende weitere Anforderungen an den Power 5:

- verbesserte Performance und
- andere server-spezifischen Anforderungen (wie Virtualisierung)[16].

Wie aus Abbildung 4.3 ersichtlich hat der Power 5 - zum Unterschied zu den anderen Prozessoren - einen L3 Cache; diesen hat es auch beim Vorgänger des Power 5 (dem Power 4) gegeben; er befand sich allerdings bei diesem zwischen dem Prozessor Chip und dem Memory Controller. Beim Power 5+ befindet sich nun auch der Memory Controller auf dem Chip, weshalb der L3 Cache auf die Prozessorseite des Chips weichen musste. Dadurch ergaben sich einige Vorteile, weil sich die Latenzzeit bei L3 Zugriffen und auch zum Memory verringerte und die Bandweite zum L3 Cache sich dadurch erhöhte.
Der L3 Cache funktioniert als “Victim Cache”, was bedeutet, dass jedes mal, wenn eine neue Cache Line in den L2 Cache geschrieben wird und dabei eine bestehende Line überschreibt, die überschriebene Line in den L3 Cache übertragen wird und erst von dort in den Hauptspeicher gelangt. Der L3 Cache wird nur auf diese Art und Weise gefüllt; Daten anderer Herkunft

werden nicht in den L3 Cache geschrieben. Nur veränderte Daten, die im L3 ersetzt werden, werden zurückgeschrieben; nicht veränderte Daten werden verworfen.

Der Power 5 Prozessor implementiert die 64-bit PowerPC Architektur von IBM (momentan sind 266 MHz DDR1 bzw. 533 MHz DDR2 DIMMS am Markt). In einem Chip befinden sich zwei identische Cores, die sich auf demselben *Die* befinden. Dem Betriebssystem präsentiert sich der Power 5 als ein 4-way symmetric Multiprozessor. Die zwei Cores auf einem *Die* teilen sich jeweils einen 1,9 MB großen L2 Cache. Dieser hat 3 Partitionen, wobei jede dieser identen Partitionen *10-way set-associative* ist und einen eigenen Controller besitzt. Über die physikalische Adresse kann bestimmt werden, in welcher Partition sich ein Cache Hit befindet. Auf die L2 Cache Controller kann unabhängig von allen Cores zugegriffen werden.
Das L3 Cache Verzeichnis befindet sich ebenfalls noch auf dem Chip, so dass nach Daten auf dem L3 Cache ohne Zeitverzögerung gesucht werden kann. Auch der L3 Cache ist nach dem selben Prinzip wie der L2 Cache implementiert. Auch er besteht aus drei Partitionen; allerdings sind diese *12-way associative* und haben eine Cache Line Länge von 256 Byte, die als zwei 128 Byte Sektoren verwaltet werden, um eine Übereinstimmung mit dem L2 Cache Lines zu erreichen.
Zur Verringerung der Latenzzeiten bei Zugriffen auf den Hauptspeicher, befindet sich beim Power 5 der Memory Controller direkt auf dem Chip.
Für jeden Lese- und Schreibzugriff auf den L2 Cache wird eine eigene Cache Kohärenz Engine gestartet, die den Status der Cache Einträge überprüft, bis die Operation zu Ende ist.
Der Zugriff auf den L3 Cache erfolgt über zwei unidirektionale 16 Byte große Datenbusse, die über die halbe Geschwindigkeit der Prozessorfrequenz verfügen. Der Zugriff auf den sich auf dem Chip befindlichen Memory Controller und den Hauptspeicher selbst erfolgt über zwei unidirektionale Datenbusse, die mit der doppelten Frequenz des in-line Memory Modules arbeiten. [16]

Bei den nachfolgenden Ergebnissen wurden aus organisatorischen Gründen immer nur drei Cores von einem Power 5 genutzt, wovon sich zwei auf demselben *Die* und einer auf dem anderen *Die* befand.
Dies war durch den Einsatz von LPARS (Dynamic Logical Partitioning) bedingt. Diese Technologie ermöglicht es, einen Großrechner in mehrere virtuelle Systeme aufzuteilen, auf denen sogar verschiedene Betriebssysteme installiert werden können. Den einzelnen Partitionen können dann verschiedene Ressourcen (wie zum Beispiel Hauptspeicher oder nur einzelne Cores des Gesamtsystemes) zugewiesen werden[9].

4.4. Intel Core 2 Extreme

Für die Tests wurde der Intel Core 2 Extreme QX6700 verwendet. Er besteht aus 2 *Dies* mit 2 Cores je *Die*; das macht in Summe 4 Cores. Daher wird der Intel Core 2 Extreme von Intel als Quad Prozessor bezeichnet, wobei es sich um einen Desktop Prozessor handelt. Dieser Prozessor bietet keine Unterstützung für Multithreading - das Betriebssystem erkennt daher auch genau die 4 physikalischen CPUs.

Der Intel Core 2 Extreme wird seit Juli 2006 zum Verkauf angeboten. Er ist von Intel als Nachfolger für den dual-core Pentium Extreme entwickelt worden. [10]

Abbildung 4.4 zeigt ein schematisches Blockdiagramm des Intel Core 2 Extremes dar. Der Intel Core 2 Extreme hat einen L1 Cache in der Größe von 32 KB per Core und einen L2 Cache von insgesamt 8 MB, d.h. es teilen sich jeweils zwei Cores auf einem *Die* 4MB L2 Cache. Die Latenzzeit bei den Zugriffen beträgt beim L1 Cache 3 Clocks, für den L2 Cache sind es 14 Clocks.

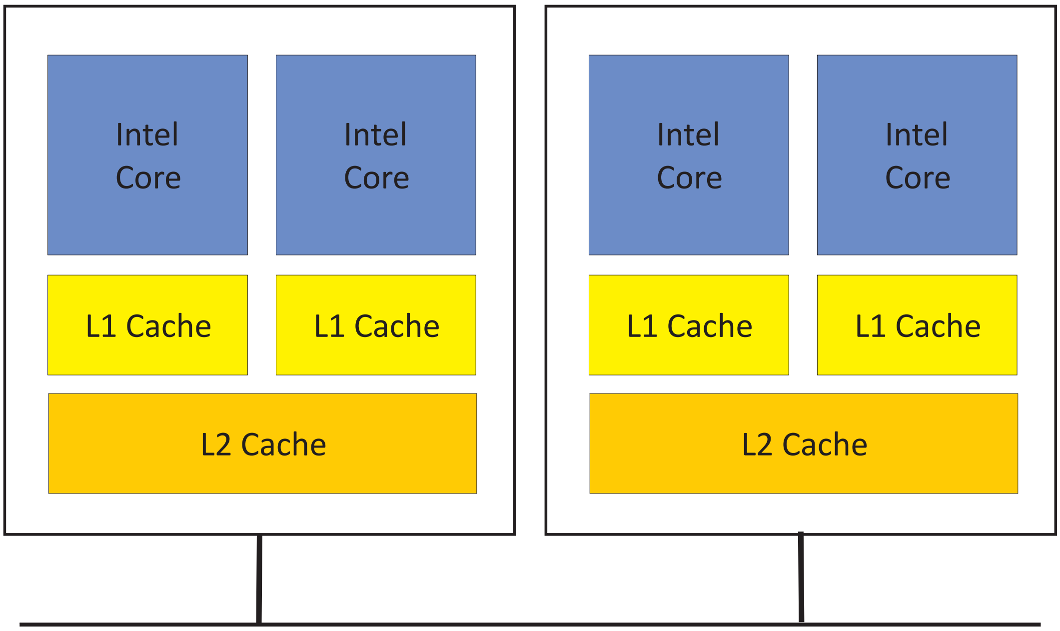


Abbildung 4.4.: Blockdiagramm Core 2 Extreme

Auf diesem Prozessor wurden die Experimente ebenfalls in C durchgeführt. Kompiliert wurde hier mit dem GNU gcc in der Version 4.1.2 Kompiler und als Betriebssystem diente Fedora Core FC6.

4.5. Vergleiche

Dieser Abschnitt gibt, in Tabelle 4.1, nochmals einen vergleichenden Überblick über die oben beschriebenen Prozessoren und die Latenzzeiten bei Zugriffen auf die verschiedenen Caches und auf den Hauptspeicher, weil diese für die Experimente von besonderer Bedeutung sind.

	Sun T1 Niagara	IBM Power5+	Intel Core2Extreme	AMD Opteron
Cores	8	4 (2 per Die)	4 (2 per Die)	2
Multithreading	fine gained	SMT	Nein	Nein
Instructions per Clock per Core	1	4	4	3
L1 D cache per Core	8 KB	32 KB	32 KB	64 KB
on-chip L2 Cache	3MB Shared	1,9 MB Shared	8 MB (4 MB per Die)	1MB per Core
off-chip L3 Cache	-	36 MB	-	-
Cache Line size in bytes L1D/L2/L3	16/64/-	128/128/256	64/64/-	64/64/-
Latency L1 D (Clocks)	1	2	3	2
Latency L2 (Clocks)	23	13	14	7
Latency L3 (Clocks)	-	87	-	-

Tabelle 4.1.: Überblick Prozessoren

5. Erstellen der Experimente

5.1. Erläuterung der Experimente

Da es sich bei Scientific Computing Anwendungen meist um Anwendungen mit großen Matrizen- oder Vektorberechnungen handelt, ist es hier von besonderer Bedeutung, wie stark durch Datenlokalität Effizienz beeinflusst wird und welche Auswirkung die Aufteilung der Berechnungen auf mehrere Threads haben kann.

Die nachfolgend im Detail erklärten Experimente sollen den maximalen Effekt zeigen, mit dem ein Best Case Szenario und ein Worst Case Szenario erreicht werden kann. Im ersten Experiment wurden synthetische Berechnungen erstellt, um die Prozessoren so zu beanspruchen, dass ein maximaler Effekt der Auswirkung von schlechter Datenlokalität erreicht werden kann.

Im zweiten Experiment geht es dann darum aufzuzeigen, wie man mit schlechter Datenlokalität umgehen kann, weil es nicht immer möglich ist, schlechte Datenlokalität zu verhindern. Bei diesem Experiment werden besonders die Eigenschaften der neuen Multiprozessorsysteme ausgenutzt.

5.1.1. Experiment 1

In Experiment geht es darum darzustellen, wie viel Performance-Gewinn durch gute Datenlokalität erreicht werden kann.

Es wird über ein großes Array iteriert und die Anzahl der Zugriffe auf dieses Array werden dabei maximiert. Auf jedem für das Experiment verwendeten Core befindet sich ein Thread. Dabei werden hauptsächlich Zuweisungen vorgenommen, während die Berechnungen möglichst gering gehalten werden. Nur einfache Integer-Berechnungen werden durchgeführt, um einen Nachteil für den Sun T1 zu vermeiden.[1]

---

[1] Der Sun T1 besitzt nur eine FPU, die sich alle Cores teilen.

Es wurden zwei extreme Varianten des Berechnungskernels für dieses Experimente erstellt: Einerseits eine Version in der es zu so vielen Cache Misses wie möglich kommt und andererseits eine Version in der die Threads sich bei ihrer Arbeit praktisch nicht stören.

Bei der zuerst genannten Variante des Berechnungskernels iterieren alle Threads, jeder auf einem anderen Core - über denselben Datenbereich des Arrays - wie in Abbildung 5.1 dargestellt - und führen hier ihre Zuweisungen und einfachen Berechnungen aus. Jedesmal, wenn ein Thread zu einem weiteren Feld des Arrays kommt, ist der Eintrag in seinem Cache vom Cache Koherenz-Protokoll bereits als ungültig markiert worden, weil ein Thread auf einem anderen Core ihn bereits wieder verändert hat. Somit kommt es praktisch jedesmal zu einem Cache Miss und es muss die Line neu in den Cache eingelesen werden. Dies benötigt Zeit und wirkt sich nachteilig auf die Laufzeit des Programmes aus.
Damit wird sozusagen das extreme worst-case Szenario von schlechter Datenlokalität nachgebildet.

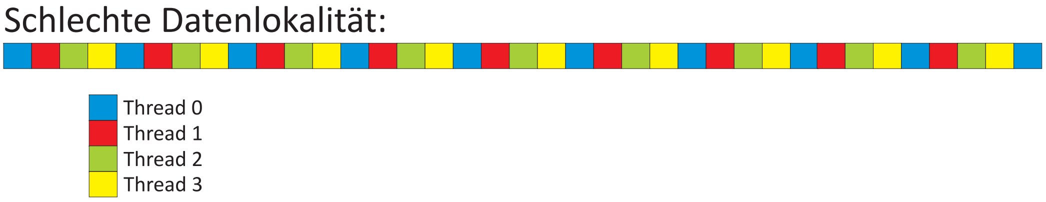


Abbildung 5.1.: Experiment 1: Schlechte Datenlokalität

Bei der Variante des Berechnungskernels, bei der gute Datenlokalität dargestellt werden soll, iterieren wieder die Threads der einzelnen Cores über einen Datenbereich eines großen globalen Arrays, nur hat jetzt jeder Thread seinen eigenen Datenbereich und kommt nicht in die Nähe der Datenbereiche der anderen Threads. So werden die Daten jedes Threads einmal in dessen Daten-Cache geschrieben und können für jede weitere Berechnung von dort geholt werden. Damit gibt es keine zeitraubenden Cache Line Transfers bei jedem einzelnen Zugriff. Wie in Abbildung 5.2 dargestellt, soll dieser Berechnungskernel darstellen, wie Threads am effizientesten eingesetzt werden können und eine optimale Leistungssteigerung erzielt wird, wenn die Datenlokalität so gewählt ist, dass die einzelnen Threads nur auf ihren lokalen Daten arbeiten und die der anderen nicht benötigen. Damit wird dargestellt, wie Datenlokalität in einem best-case Szenario aussehen sollte.

Da sich die meisten Berechnungen nicht auf eines der beiden oben beschriebenen Extreme anpassen lassen, werden auch in den Experimenten Zwischenversionen verwendet, um einen

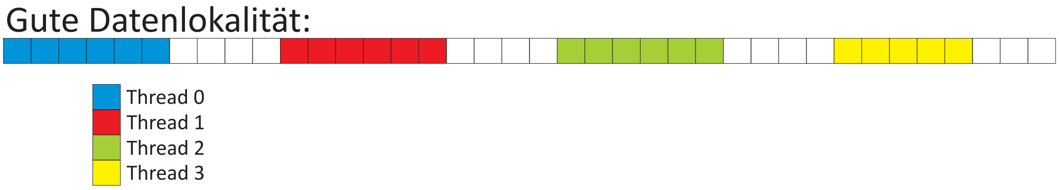


Abbildung 5.2.: Experiment 1: Gute Datenlokalität

besseren Übergang darstellen zu können. Je schlechter die Datenlokalität in den Experimenten ist, desto öfter kommt es beim Zugriff auf eine Variable zu Cache Misses.

Durch die Darstellung des maximalen Effekts, der mit guter Datenlokalität erreicht werden kann, soll die Bedeutung von Datenlokalität ermittelt werden. Anhand dieser Messungen soll klar werden, welche Auswirkungen gute Datenlokalität auf die Laufzeit eines Programmes haben kann.

5.1.2. Experiment 2

Da es nicht immer einfach ist, eine so gute Datenlokalität zu erreichen (wie in dem künstlichen Experiment 1 im Fall der guten Datenlokalität), wird nun im Experiment 2 nach einer Möglichkeit gesucht, wie man trotzdem eine gute Laufzeit erzielen kann. Wie in Abbildung 5.3 dargestellt soll dies durch das Aufteilen der Arbeit von einem einzelnen Thread auf mehrere erreicht werden, denn so können die Stalls der Threads, die Cache Misses haben, durch das Ausführen eines anderen Threads genutzt werden. Besonders interessant ist hier der Effekt auf den Prozessoren, die Multithreading unterstützen und dadurch - ohne Zeitverlust - zwischen den Threads wechseln können.
Bei der Sun Niagara T1 und bei der IBM Power 5+ können mehrere Threads auf einen Core mit der Absicht zugewiesen werden, dass diese - während ein anderer Thread nach einem Cache Miss auf einen Cache Line transfer wartet - ausgeführt werden.

Das Experiment ist folgendermaßen aufgebaut:

- Begonnen wird, indem immer nur ein Thread auf einen Core zugewiesen wird; dies entspricht genau der Teststellung, die in Experiment 1 verwendet wurde - das Experiment wird jeweils für die beiden Extremfälle (gute und schlechte Datenlokalität) durchgeführt;

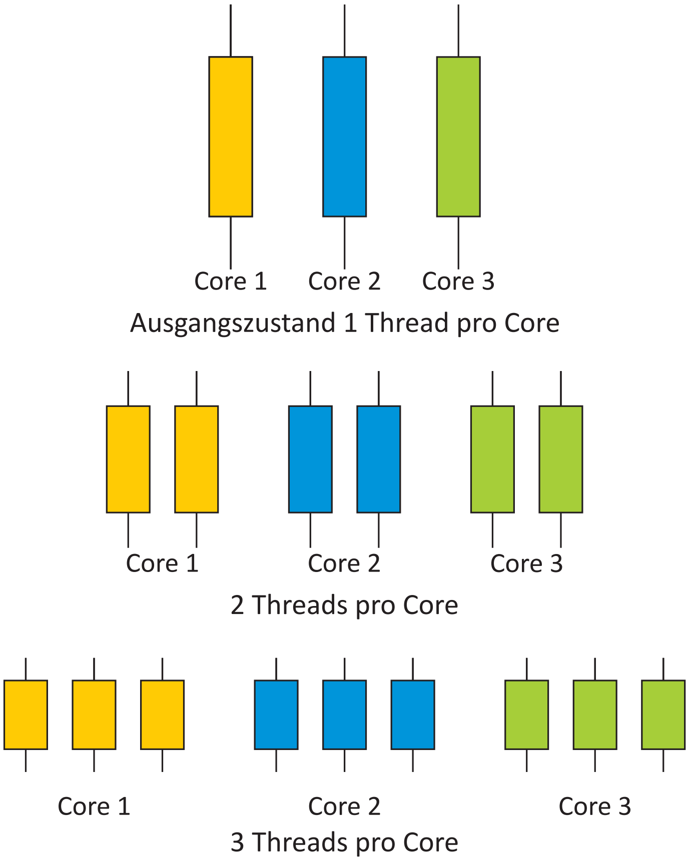


Abbildung 5.3.: Experiment 2: Schematische Darstellung des Testaufbaus

- Weitergeführt wird dieses Experiment so, dass die Arbeit von jedem Thread auf zwei Threads aufgeteilt wird (jeweils zwei Threads, die sich eine Arbeit geteilt haben, werden auf denselben Core zugewiesen).
- Bei der nächsten Teststellung wird die Arbeit gedrittelt und so auf die drei Threads verteilt, dass diese ebenfalls auf denselben Core zugewiesen werden.
- Schliesslich geht es mit vier Threads pro Core weiter (darauf folgen fünf Threads, sechs Threads usw.).

In diesem Experiment sind die unterschiedlichen Hardwareplattformen von größerer Bedeutung als beim Experiment 1. Eigentlich ist nur die Sun T1 ein “echter” Multicore, weil sich nur bei diesem Prozessor alle Cores auf demselben *Die* befinden. Die IBM Power 5+ und der Intel Core 2 Extreme bieten jeweils zwei Cores auf zwei *Dies* an, die sich in dem selben Quad-Core Package befinden. Bei diesen beiden Hardware Plattformen stellt sich die Frage, ob es eine Bedeutung hat, auf welchem *Die* sich die Cores befinden auf welche die Threads zugewiesen werden.

5.2. Pseudo-Code der Experimente

5.2.1. Experiment 1

Für Experiment 1 sind hier die beiden extremen Varianten angeführt - gute und schlechte Datenlokalität. Bei diesem Experiment arbeitet nur ein Thread pro Core.

In Listing 5.1 ist die Erzeugung eines Threads mit einem Berechnungskernel mit schlechter Datenlokalität dargestellt. Zuerst wird der Thread auf die ihm zugeordnete CPU gebunden und danach müssen alle Threads eine Barriere durchlaufen, damit sie alle zur gleichen Zeit mit den Zugriffen beginnen. Dabei gibt *cpu-nr* die CPU Id an, die das Betriebssystm erkennt und auf die der Thread gebunden werden soll; *#cores* gibt an, wieviele Cores bei der Berechnung mit Threads belegt werden. Bei diesem Berechnungskernel greifen alle Cores immer auf Felder die neben Feldern, eines anderen Threads liegen, womit es bei jedem Zugriff auf das globale Array zu einem Cache Miss kommen sollte. Die Variable *coreNr* gibt an auf welchem Core des Systems der Thread läuft; diese Nummer unterscheidet sich bei den beiden Prozessoren, die kein Multithreading unterstützen, nicht von der *cpu-nr*. Im Fall des IBM Power 5+ und der Sun T1 unterscheidet es sich allerdings schon, denn hier wird jeder Hardware Thread als eigene CPU Id vom Betriebssystem erkannt, womit das Betriebssystem so viele CPUs anzeigt, als es Hardware Threads auf dem System gibt.

```
//Procedure eines Threads
procedure neuer_Thread(cpu-nr, #cores){

  bindThreadOn(cpu-nr);

  BARRIER();

  start_time();
  //Eigentlicher Berechnungskernel
  for(i=low+coreNr, high, #cores){
    a[i]=a[i+1];
  }
  end_time();

}
```

Listing 5.1: Experiment 1: schlechte Datenlokalität

In Listing 5.1 wird in Zeile 4 der Thread auf eine CPU gebunden; die Zahl der CPU Id entspricht derjenigen, die das Betriebssystem erkennt. Danach muss jeder Thread an eine Barriere kommen, welche dazu dient, dass alle Threads zum gleichen Zeitpunkt mit ihren Berechnungen beginnen.
Vor dem eigentlichen Berechnungskernel in Zeile 8 beginnt die Zeitmessung. Danach durchläuft jeder Thread das Array und greift auf jedes *n-te* Feld zu, wobei *n* die Anzahl der verwendeten Cores wiedergibt (bei Experiment 1 wird immer nur ein Thread pro Core verwendet). Die eigentliche Berechnung findet auf einem global definierten Integer Array *a* statt - wobei hier die Zugriffe maximiert werden und sehr wenige Berechnungen durchgeführt werden. Dies geschieht indem hauptsächlich Umspeicherungen der Felderinhalte vorgenommen werden.

In Listing 5.2 wird abermals Experiment 1 dargestellt - hier allerdings mit guter Datenlokalität. Die einzelnen Threads iterieren über einen bestimmten Block des Arrays, dieser wird anhand ihrer CPU Id Nummer bestimmt, wobei während des Zugriffs immer auf benachbarte Felder des Array zugegriffen wird; kein anderer Thread benötigt Daten aus diesen Feldern und somit sollte es zu keinen Cache Misses kommen.

```
//Procedure eines Threads
procedure neuer_Thread(cpu-nr, #cores){

  bindThreadOn(cpu-nr);

  BARRIER();

  start_time();
  //Eigentlicher Berechnungskernel
  for(i=low[coreNr], high[coreNr], i++){
    a[i]=a[i+1];
  }
  end_time();

}
```

Listing 5.2: Experiment 1: gute Datenlokalität

In Listing 5.2 findet das Vorbereiten des Threads genauso statt, wie es bei schlechter Datenlokalität der Fall wäre. Der einzige entscheidende Unterschied ist hier die Arbeit auf dem globalen Array a die einzelnen Threads greifen nur blockweise jeweils auf das Array zu und besuchen dabei jedes Feld.

5.2.2. Experiment 2

In Listing 5.3 ist die Procedure für einen Thread angegeben. Der Unterschied zu Experiment 1 ist hier nur, dass die Arbeit durch die Anzahl der Threads geteilt wird - in diesem Beispiel eben durch drei. Wobei *ThreadNR* die Threads auf dem jeweiligen Core hoch zählt und nichts über die Gesamtanzahl der Threads, die das Array bearbeiten, aussagt.

```
//Procedure eines Threads
procedure neuer_Thread(cpu-nr, #cores){

  bindThreadOn(cpu-nr);

  BARRIER();

  start_time();
  //Eigentlicher Berechnungskernel
  for(i=low[coreNr]+ThreadNR, high[coreNr]/3, i+=#cores+ThreadNR){
    a[i]=a[i+1];
  }
  end_time();
}
```

Listing 5.3: Experiment 2: schlechte Datenlokalität

In den Listings 5.4 bis 5.6 wird die Erstellung von drei Threads für gute Datenlokalität angegeben, die sich die Arbeit von einem Core teilen sollen.

Hier iterieren die Threads über einen eigenen Bereich des Arrays und besuchen immer nur benachbarte Felder, auf die kein anderer Thread zugreifen muss. Allerdings wird auch hier - wie in Experiment 2 - mit schlechter Datenlokalität die Arbeit auf die drei Threads pro Core aufgeteilt. Im Fall der guten Datenlokalität geschieht dies in Blockform, jeder der Threads hat einen eigenen Teil des Block zu bearbeiten.

```
//Procedure eines Threads
procedure neuer_Thread(cpu−nr, #cores){

  bindThreadOn(cpu−nr);

  BARRIER();

  start_time();
  //Eigentlicher Berechnungskernel
  for(i=low[coreNr], high[coreNr]/3, i++){
    a[i]=a[i+1];
  }
  end_time();

}
```

Listing 5.4: Experiment 2: gute Datenlokalität Thread 1 auf Core X

```
//Procedure eines Threads
procedure neuer_Thread(cpu−nr, #cores){

  bindThreadOn(cpu−nr);

  BARRIER();

  start_time();
  //Eigentlicher Berechnungskernel
  for(i=high[coreNr]/3, high[coreNr]∗2/3, i++){
    a[i]=a[i+1];
  }
  end_time();

}
```

Listing 5.5: Experiment 2: gute Datenlokalität Thread 2 auf Core X

```
//Procedure eines Threads
procedure neuer_Thread(cpu−nr, #cores){

  bindThreadOn(cpu−nr);

  BARRIER();

  start_time();
  //Eigentlicher Berechnungskernel
  for(i=high[coreNr]∗2/3, high[coreNr], i++){
    a[i]=a[i+1];
  }
  end_time();

}
```

Listing 5.6: Experiment 2: gute Datenlokalität Thread 3 auf Core X

Würde man nun diese Berechnung für *n* Threads pro Core ausführen, so würden sich die einzelnen Blockstücke folgendermassen berechnen:
Um es übersichtlicher zu gestalten wurde bei den folgenden Berechnungen bei den Variablen *low* und *high* jeweils die *cpu-nr* weggelassen. Diese gibt jeweils den Bereich an, den ein einzelner Core zu bearbeiten hat; wie das zu geschehen hat, ist bei der Arbeitsaufteilung auf die einzelnen Cores nicht wirklich relevant.

Thread 1:	low	bis	$\frac{high}{n} - 1$
Thread 2:	$\frac{high}{n}$	bis	$\frac{2*high}{n} - 1$
Thread 3:	$\frac{2*high}{n}$	bis	$\frac{3*high}{n} - 1$
	...		
Thread n-1:	$\frac{(n-2)*high}{n}$	bis	$\frac{(n-1)*high}{n} - 1$
Thread n:	$\frac{(n-1)*high}{n}$	bis	$high$

Um diese beiden Experimente sinnvoll durchführen zu können, sind zwei besondere Schritte bei der Implementierung nötig.
Einerseits muss der Light Weight Prozess an eine bestimmte CPU gebunden werden. Dies geschieht mittels einen Betriebssystemaufrufs (in diesen Experimenten in C). Je nach Betriebssystem sieht der Funktionsaufruf anders aus.
Andererseits gibt es keine Funktion in Java, welche die Möglichkeit bietet, einen Thread an eine CPU zu binden. Daher muss auf das Java Native Interface zurückgegriffen werden, denn erst dieses ermöglicht einen Methodenaufruf in einer anderen Sprache.
Diese beiden Schritte werden in den folgenden beiden Abschnitten noch im Detail erläutert.

5.2.3. Processor Bind

Sun Solaris Bei dem Betriebssystem Solaris erfolgt der Aufruf zum Binden eines einzelnen Threads an eine CPU mit der Methode:

```
int processor_bind(idtype_t idtype, id_t id,
  processorid_t processorid, processorid_t *obind)
```

Wobei *idtype* angibt, ob es sich um einen Light Weight Prozess oder um einen gesamten Prozess handelt (P_ID für eine Prozess Id und P_LWP für eine Light Weight Prozess id). Die eigentliche Identifikationsnummer des Prozesses oder des Light Weight Prozesses wird in *id* festgelegt. In den oben beschriebenen Experimenten wurde jeweils die erste Variante verwendet. Dabei wurde ein Thread - ein Light Weight Prozess - an eine bestimmte CPU gebunden. Wird hingegen ein Prozess an eine CPU gebunden, so werden alle seine Threads ebenso an

diese CPU gebunden.
Mit *processorid* wird die Id des Prozessors angegeben, an die der Light Weight Prozess gebunden werden soll. Diese Id bezieht sich auf die Id, die das Betriebssystem den CPUs gibt (z.B. bei der Sun Niagara T1 erhält jeder Strand eine CPU Id vom Betriebssystem). Wird der Befehl erfolgreich ausgeführt, so gibt der Methodenaufruf 0 zurück[18].

Linux - Fedora Core Unter den meisten Linux Derivaten wird ein Thread oder Prozess mittels

```
int sched_setaffinity(pid_t pid, unsigned int len,
   unsigned long *mask);
```

an eine CPU gebunden.
Mit *pid* wird die Prozess oder Light Weight Prozess Id angegeben, wird hier 0 eingegeben so wird als *pid* immer der aufrufende Prozess eingesetzt; mit *mask* wird das CPUSet angegeben auf das der Prozess zugewiesen werden kann. Wenn es sich nur um eine einzige CPU handeln, so wird es ebenfalls in den Experimenten verwendet; unsigned int len gibt die Größe der darauf folgenden mask an, in dem das CPUSet enthalten ist[14].

IBM AIX Bei der Entwicklung unter IBM AIX wird ein Prozess oder Thread mittels

```
int bindprocessor(int what, int who, cpu_t where)
```

durchgeführt. Dieser Befehl bindet einen Thread oder einen gesamten Prozess auf eine CPU Id; andere Threads, die sich auf der CPU gerade in Ausführung befinden, werden sofort vom Scheduler auf eine andere CPU migriert.
Analog zum processor_bind unter Solaris, gibt der erste Parameter an, ob es sich um einen ganzen Prozess oder um einen einzelnen Thread handelt. Der zweite Parameter gibt die genaue Id des Threads oder des Prozesses an, während der letzte Parameter angibt, auf welche CPU der Thread oder der Prozess hingebunden werden soll.
Im Fehlerfall wird -1 zurückgegeben, bei erfolgreicher Durchführung 0[8] .

5.2.4. Das Java Native Interface

Auf der Sun Niagara T1 wurden die beiden Experimente nicht nur in C sondern auch in Java ausgeführt. Da es sich bei Java um eine betriebssystemunabhängige Sprache handelt, gibt es in

Java selbst keine Möglichkeit Betriebssystemroutinen, wie zum Beispiel das Zuweisen eines einzelnen Threads auf eine CPU Id des Betriebssystemes, aufzurufen.
Da aber nicht einmal das Bereitstellen der gesamten Java Standard API, wie sie heute existiert, möglich wäre ohne das Verwenden von betriebssystemspezifischen Funktionen (wie z. B. bei den meisten I/O Funktionen), bietet Java ein standardisiertes Interface an, das es ermöglicht, zwischen Java und einer betriebssystemspezifischen Sprache eine Verbindung herzustellen; dieses Interface wird als Java Native Interface (JNI) bezeichnet.

Das JNI erlaubt den Aufruf von Methoden, die in einer Java Klasse als *native* deklariert sind. Diese enthalten nur die Signatur der Methode, die Methode selbst wird in der „nativen“ Programmiersprache ausprogrammiert.
Dieses Interface ist nicht nur für den Programmierer, der auf Rechnerressourcen zugreifen will, von Nutzen sondern es ist auch nötig, um die gesamte Standard Java API zu erstellen. Dies ist deshalb der Fall, weil die meisten Methoden, die sich mit der Darstellung von Objekten auf dem Bildschirm oder dem Zugreifen auf bestimmte Dateien beschäftigen, Methoden sind, die in einer Programmiersprache erstellt wurden, die Zugriffe auf Betriebssystemfunktionen erlaubt[11].

In unseren Experimenten wurde die native Methode in C ausprogrammiert. Konkret handelt es sich dabei um die Methode, mit welcher der Thread selbst an eine CPU gebunden wird. Diese Methode wird in der Klasse wie eine abstrakte Methode deklariert (es wird nur die Signatur deklariert, während der Körper der Methode keine Implementierung erhält)[19].

```
public native void bind(int cpu);
```

Nach der erfolgreichen Implementierung der Java Klasse wird die Datei nun kompiliert. Danach wird mit dem *javah* Generator, der standardmäßig bei jeder JDK Installation mitgeliefert wird eine C/C++ Header Datei erzeugt, die in C/C++ implementiert werden muss. In dieser Header Datei behält die Methode im Grunde ihre ursprüngliche Signatur, allerdings wird noch das Package der Klasse miteinbezogen, um eine eindeutige Zuordnung einer nativen Methode zu einer Klasse erreichen zu können. [11]

Eine solche Header Datei sieht folgendermaßen aus:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class multicoretest_common_BindThread */

#ifndef _Included_multicoretest_common_BindThread
#define _Included_multicoretest_common_BindThread
#ifdef __cplusplus
extern "C" {
```

```
#endif
#undef multicoretest_common_BindThread_MIN_PRIORITY
#define multicoretest_common_BindThread_MIN_PRIORITY 1L
#undef multicoretest_common_BindThread_NORM_PRIORITY
#define multicoretest_common_BindThread_NORM_PRIORITY 5L
#undef multicoretest_common_BindThread_MAX_PRIORITY
#define multicoretest_common_BindThread_MAX_PRIORITY 10L
/*
 * Class:     multicoretest_common_BindThread
 * Method:    bind
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_multicoretest_common_BindThread_bind
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Nun zur Implementierung für die Sun Niagara T1. Hier wird der Leight Weight Process mittels processor_bind an eine bestimmte CPU gebunden. Die Art der Methode für das Zuweisen eines Threads an eine CPU ist - je nach Betriebssystem - verschieden; dies wird in Abschnitt 5.2.3 erläutert.

```
#include <pthread.h>
#include <stdio.h>

#include <unistd.h>
#include <sys/processor.h>
#include <sys/procset.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/procfs.h>
#include <sys/lwp.h>

#include "multicoretest_common_BindThread.h"

JNIEXPORT void JNICALL
  Java_multicoretest_common_BindThread_bind(JNIEnv *env, jclass c, jint cpu)
{
        int cpuc = (int)cpu;
        int rc;
        rc = processor_bind(P_LWPID, P_MYID, cpuc, NULL);
}
```

Nach der Implementierung der Methode in C wird der Code mit der Verlinkung auf das entsprechende Interface kompiliert.

Das nachfolgende Beispiel ist wieder für die Solaris Umgebung auf der Sun Niagara T1 gedacht, jedoch könnte mit dem GNU gcc Kompiler der Befehl (analog) ebenso erfolgen.

```
cc -G -I/usr/java/include
  -I/usr/java/include/solaris cpu.c -o bindThread.so
```

Der letzte Schritt, um die native Methode erfolgreich aufzurufen, besteht nur noch darin, die mit dem oben angeführten Befehl erstellte Library während der Laufzeit in die Java Applikation zu laden. Bei den Experimenten wird dies ganz zu Beginn, nach dem Abfragen der Startparameter, gemacht. Dies geschieht mit der Methode *System.loadLibrary(„/path/to/bindThread.so“);*. Es gibt aber auch andere Methoden, mit denen man eine Library laden kann. Diese unterscheiden sich eigentlich nur darin, an welchem Ort im Filesystem die Library abgelegt wurde. Befindet sich die Library nicht auf dem angegebenen Pfad und ist der JVM somit nicht verfügbar, kommt es zu einer *RuntimeException* einer *java.lang.UnsatisfiedLinkErrorException*. Diese wird bereits beim Laden der Library und nicht erst beim ersten Aufruf der Methode, die diese Library verwendet, geworfen[11].

Die Verwendung des JNI ist noch in eine andere Richtung möglich; es kann aus einer anderen Sprache - beispielsweise C/C++ - heraus Java Methoden ansprechen. Darauf wird allerdings in dieser Section nicht weiter eingegangen, weil es für die Erstellung der Experimente nicht erforderlich ist.

6. Durchführen der Experimente

6.1. Sun UltraSPARC T1 Niagara

Experiment 1 Die nachfolgenden Diagramme enthalten normalisierte Werte der gemessenen Ergebnisse. Die längste Ausführungzeit wurde jeweils auf 1 gesetzt; dies soll einen einfacheren Vergleich zwischen den einzelnen Diagrammen und somit zwischen den einzelnen Rechnerarchitekturen ermöglichen.

Abbildung 6.1 zeigt die Laufzeiten für das erste Experiment auf der Sun T1. Dabei gibt die x-Achse den Grad der Datenlokalität an. Auf der linken Seite wird mit schlechter Datenlokalität begonnen; hier arbeiten alle Threads auf demselben Bereich des Arrays, was zu einer maximalen Anzahl an Write Misses und damit jedesmal zu einem Cache Line Transfer zwischen den Cores führt. Auf der rechten Seite ist gute Datenlokalität eingetragen; hier arbeiten die Threads auf verschiedenen Teilen des Arrays und weil kein anderer Thread auf diese Bereiche zugreifen kann (somit kann es zu keinen Cache Konflikten zwischen den Cores kommen), sind auch keine Cache Line Transfers nötig. Bei diesem Experiment wird pro Core genau ein Thread zugewiesen, wobei jeder dieser Threads genau die gleiche Arbeit zu verrichten hat. Der einzige Unterschied ist - je nach der angegebenen Datenlokalität - darin zu finden, auf welchem Bereich des globalen Arrays diese Arbeit verrichtet wird.

Im Gegensatz zu den anderen verwendeten Prozessoren hat die Sun Niagara T1 eine einheitliche Struktur, bei der sich alle Cores auf demselben Die befinden. Daher musste auf die Verteilung der Threads auf die Cores keine weitere Rücksicht genommen werden.
Es wurden sechs Threads auf alle sechs Cores verteilt. Durch die Verbesserung der Datenlokalität konnte eine Beschleunigung um den Faktor von 1,60 in C und von 1,50 in Java bei der Ausführungszeit erreicht werden.

In Tabelle 6.1 und 6.2 werden die normalisierten Werte der Messungen und deren Speedup dargestellt. Der Wert der Lokalität bezieht sich auf in 5.2 erläuterten LINESZ. Die dort ange-

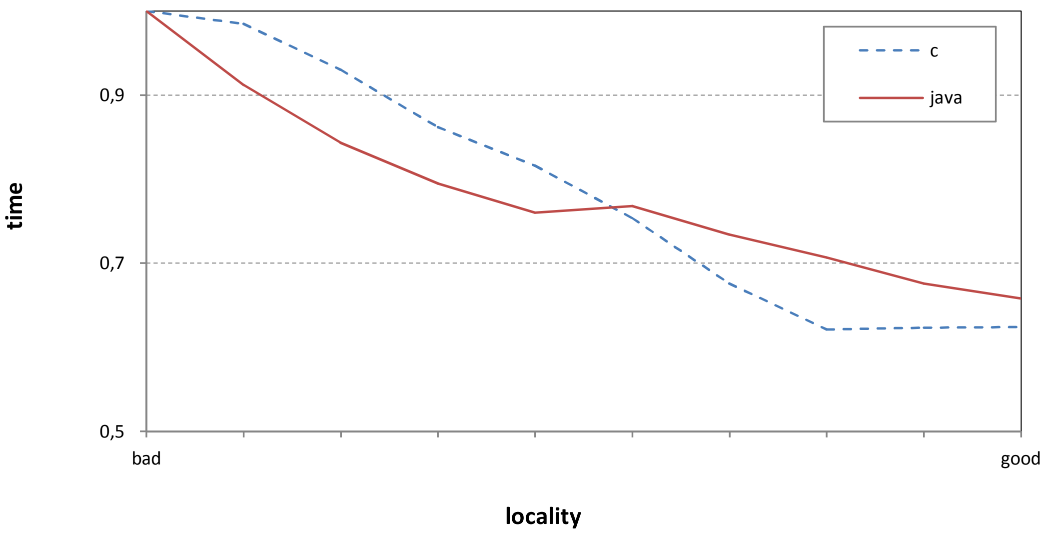


Abbildung 6.1.: Sun Niagara T1 Experiment 1

führten Werte entsprechen jenen, die tatsächlich in den einzelnen Messungen im Code angeführt waren.

In Abbildung 6.2 wurden in Java zuerst zwei Threads auf zwei Cores, dann vier Threads auf vier Cores und schließlich sechs Threads auf sechs Cores gemessen. Bei dieser Grafik ist zu beachten, dass der Arbeitsaufwand - je nach Anzahl der Cores - unterschiedlich groß ist, denn jeder Thread verrichtet dieselbe Arbeit. Wenn es nur zwei Threads sind, so erfordert es auch nur ein Drittel der Arbeit, die bei sechs Threads (auf sechs Cores) notwendig wäre.
Dadurch kommt deutlich zum Ausdruck, dass der Einfluss von Datenlokalität mit der Anzahl der beteiligten Threads steigt. Der Speedup beträgt bei sechs Threads 1,50 während der Speedup bei zwei Threads nur 1,10 beträgt. In Tabelle 6.3 befinden sich die auf die längste Ausführungszeit normalisierten Werte der Messungen.

Experiment 2 In Abbildung 6.3 werden die Ergebnisse des zweiten Experiments dargestellt. Es wurden bei dieser Messung wieder alle sechs Cores verwendet. Dabei wird genau die gleiche Berechnung verwendet - wie beim ersten Experiment (mit der guten und schlechten Datenlokalität), nur dass diesmal zwei Durchläufe des Experimentes erfolgen und die Datenlokalität bei diesen Durchläufen konstant gehalten wird; ein Durchlauf mit schlechter und ein Durchlauf mit guter Datenlokalität. Bei jedem dieser Durchläufe wird die Arbeit, die ein Thread aus Experiment 1 verrichtet, auf mehrere Threads aufgeteilt. Diese laufen auf demselben Core, um jeweils das *Stallen* aufgrund eines Cache Misses durch das Weiterarbeiten eines

Lokalität	6 Cores	Speedup
1	1	1
2	0,98	1,01
3	0,93	1,07
4	0,86	1,16
5	0,81	1,23
6	0,75	1,33
10	0,67	1,47
12	0,62	1,60
40	0,62	1,60
156	0,62	1,60

Tabelle 6.1.: Sun T1 Experiment 1 auf 6 Cores in C, normalisierte Werte

Lokalität	6 Cores	Speedup
1	1	1
2	0,91	1,09
3	0,84	1,18
4	0,79	1,25
5	0,76	1,31
6	0,76	1,30
10	0,73	1,36
12	0,70	1,41
40	0,67	1,47
156	0,65	1,52

Tabelle 6.2.: Sun T1 Experiment 1 auf 6 Cores in Java, normalisierte Werte

anderen Threads zu überbrücken.
Die Sun Niagara T1 und die IBM Power 5+ sind für diese Aufgabe weitaus besser geeignet, weil sie hardwareseitig eine Unterstützung für das Zuweisen mehrerer Threads auf einen Core bieten.
Für jede Prozessorarchitektur gibt es nun zwei Diagramme: eines für schlechte Datenlokalität und eines für gute Datenlokalität; der Grad der Datenlokalität wird während der Durchführung eines Experimentes nicht verändert.

Auf der x-Achse wird die Anzahl der Threads pro Core aufgetragen. Auf der y-Achse wird - genau wie in Experiment 1 - die normalisierte Zeit aufgetragen; dabei wird die längste Laufzeit auf eins gesetzt; alle anderen werden daran angepasst.

Auffällig bei allen Durchläufen in Abbildung 6.3 ist, dass immer dann, wenn die Anzahl der

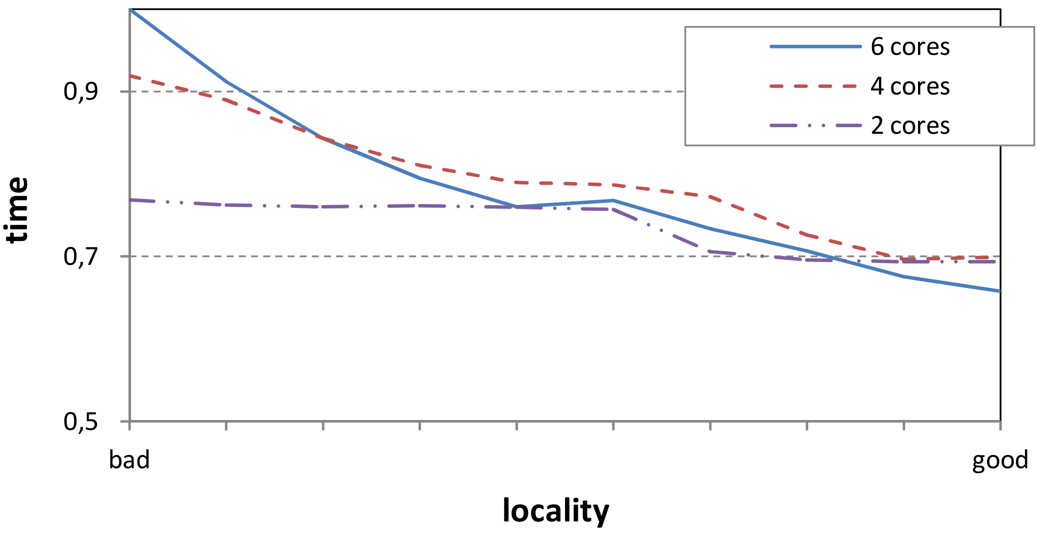
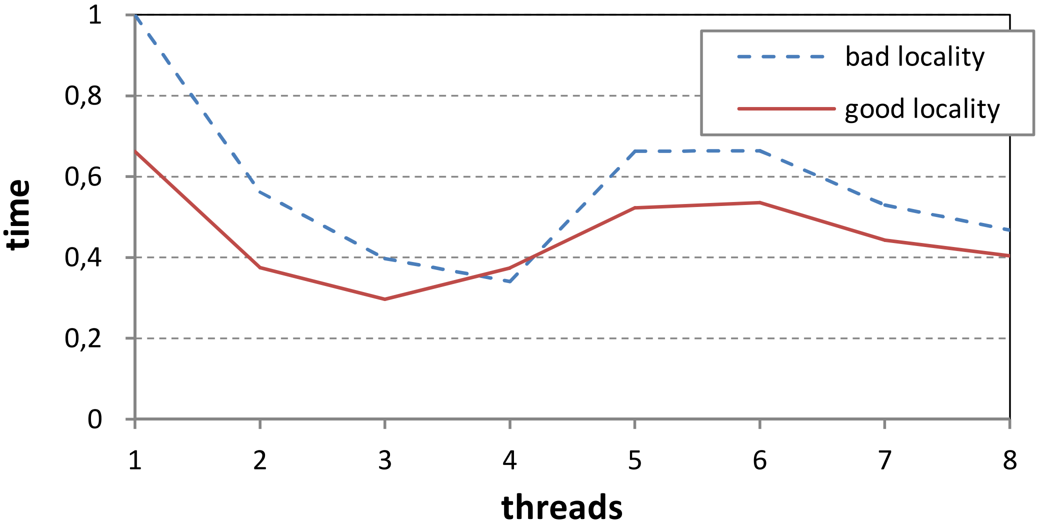


Abbildung 6.2.: Sun T1 Experiment 1

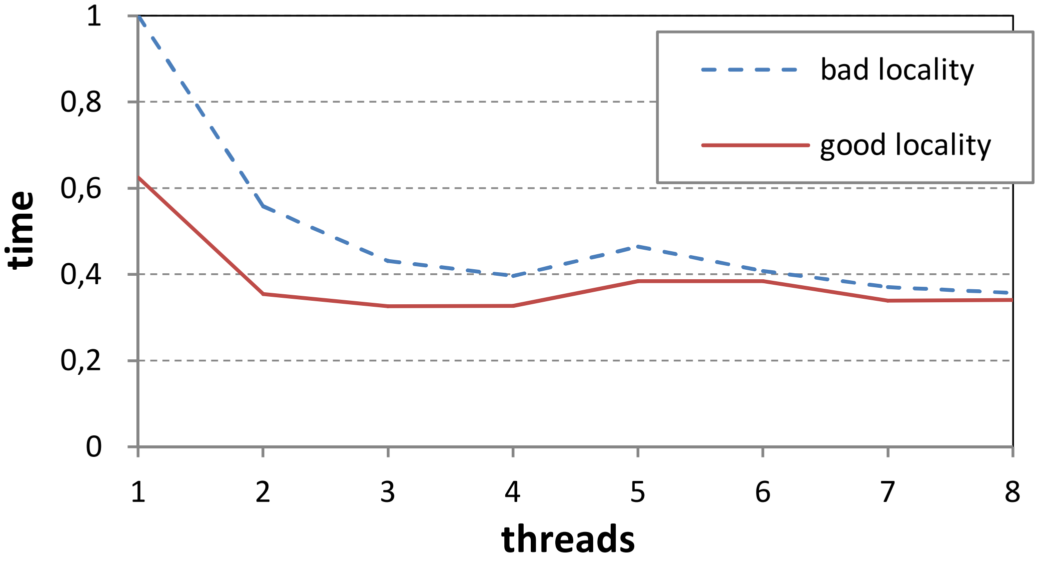
Threads vier überschreitet, die Ausführungszeit größer wird. Vermutlich liegt dies an der der Hardwareunterstützung für vier Threads pro Core, welche die Sun Niagara T1 bietet. Für den fünften Thread muss die Rechenzeit für das Migrieren der Threads verbraucht werden. Grundsätzlich ist zu bemerken, dass der Effekt bei schlechter Datenlokalität sehr viel stärker ist als bei guter Datenlokalität. Vermutlich liegt es daran, dass bei guter Datenlokalität praktisch kein Thread blockiert, weshalb das Wechseln der einzelnen Threads keinen Vorteil bringt. Bei schlechter Datenlokalität kommt es zu dauernden Blockieren eines Threads, weshalb es vorteilhaft ist, wenn ein anderer Thread - der bereit ist - die CPU weiterhin auslasten kann, während der blockierende Thread erst nach dem erfolgreichen Cache Line Transfer wieder in die Warteschlange aufgenommen wird. Bis zum vierten zusätzlichen Thread pro Core wird kein einziger Taktzyklus für das Wechseln zwischen den Threads verbraucht, womit auch in diesem Bereich der Speedup am größten ist.

In Tabelle 6.4 sind die Werte - auf die längste Ausführungszeit normalisiert - für die Messungen in Java für gute und schlechte Datenlokalität und der dazugehörige Speedup angegeben.

In Tabelle 6.5 sind die normalisierten Werte für gute und schlechte Datenlokalität in C zu finden.



(a) Java, normalisiert auf die längste Ausführungszeit



(b) C, normalisiert auf die längste Ausführungszeit

Abbildung 6.3.: Sun Niagara T1 Experiment 2

Lokalität	6 Cores	Speedup	4 Cores	Speedup	2 Cores	Speedup
1	1	1	0,92	1	0,76	1
2	0,91	1,09	0,89	1,03	0,77	1,01
3	0,84	1,18	0,84	1,09	0,77	1,01
4	0,79	1,25	0,81	1,13	0,77	1,01
5	0,76	1,31	0,79	1,16	0,77	1,01
6	0,76	1,30	0,78	1,16	0,76	1,02
10	0,73	1,36	0,77	1,18	0,71	1,08
12	0,71	1,41	0,72	1,26	0,69	1,10
40	0,67	1,47	0,69	1,31	0,69	1,10
156	0,65	1,52	0,69	1,31	0,69	1,10

Tabelle 6.3.: Sun T1 Experiment 1: 2,4,6 Cores - Java

Threads pro Core	Schlechte DL	Speedup	Gute DL	Speedup
1	1	1	0,66	1
2	0,56	1,78	0,37	1,76
3	0,39	2,51	0,29	2,22
4	0,34	2,93	0,37	1,76
5	0,66	1,50	0,52	1,26
6	0,66	1,50	0,53	1,23
7	0,52	1,88	0,44	1,49
8	0,46	2,13	0,40	1,63

Tabelle 6.4.: Experiment 2: Sun Niagara T1 in Java

6.2. IBM Power 5+

Beim IBM Power 5+ ist es wesentlich, auf welche Cores die Threads bei den einzelnen Messungen gelegt wurden. Aus technischen Gründen stehen hier zwei Cores auf einem *Die* und ein Core auf dem anderen *Die* zur Verfügung. Diesmal wurden die Experimente nur mehr in C durchgeführt.

Experiment 1 Von besonderem Interesse ist es, ob der Performance Unterschied nicht nur im Bezug auf die Datenlokalität sondern auch hinsichtlich der Cores, auf denen die Threads gelegt werden, sich auf demselben oder auf unterschiedlichen *Dies* befinden.

Durch die Verbesserung der Datenlokalität kann bei zwei Threads auf zwei Cores, die sich auf demselben *Die* befinden, ein Speedup von 1,6 erzielt werden. Bei zwei Threads auf zwei Cores, die sich auf unterschiedlichen *Dies* befinden, wird ein Speedup von 3,1 erreicht. Werden nun alle drei Cores mit jeweils einem Thread belegt, so kann durch Verbesserung der

Threads pro Core	Schlechte DL	Speedup	Gute DL	Speedup
1	1	1	0,62	1
2	0,55	1,79	0,35	1,76
3	0,43	2,31	0,32	1,91
4	0,39	2,52	0,32	1,91
5	0,46	2,15	0,38	1,62
6	0,40	2,45	0,38	1,62
7	0,37	2,70	0,33	1,84
8	0,35	2,80	0,34	1,83

Tabelle 6.5.: Experiment 2: Sun Niagara T1 in C

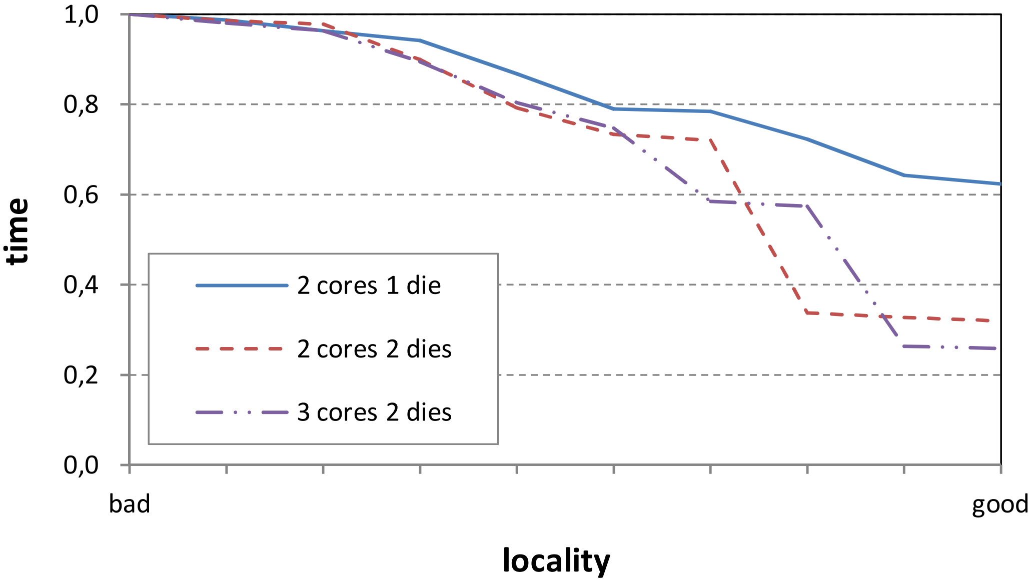
Datenlokalität ein Speedup von 3,9 erzielt werden. Daraus ergibt sich, dass Threads, die auf demselben Core auf einen gemeinsamen Datenbereich zugreifen, effizienter arbeiten als wenn sich die Cores auf unterschiedlichen *Dies* befinden.

Die Abbildung 6.4 (a) dargestellten Ergebnisse sind voneinander unabhängig, jeweils auf den höchsten Wert normalisiert worden; in Abbildung 6.4 (b) sind alle auf die längste Ausführungszeit normalisiert.
Die dazugehörigen Messergebnisse, normalisiert auf die längste Ausführungszeit, befinden sich in Tabelle 6.6.

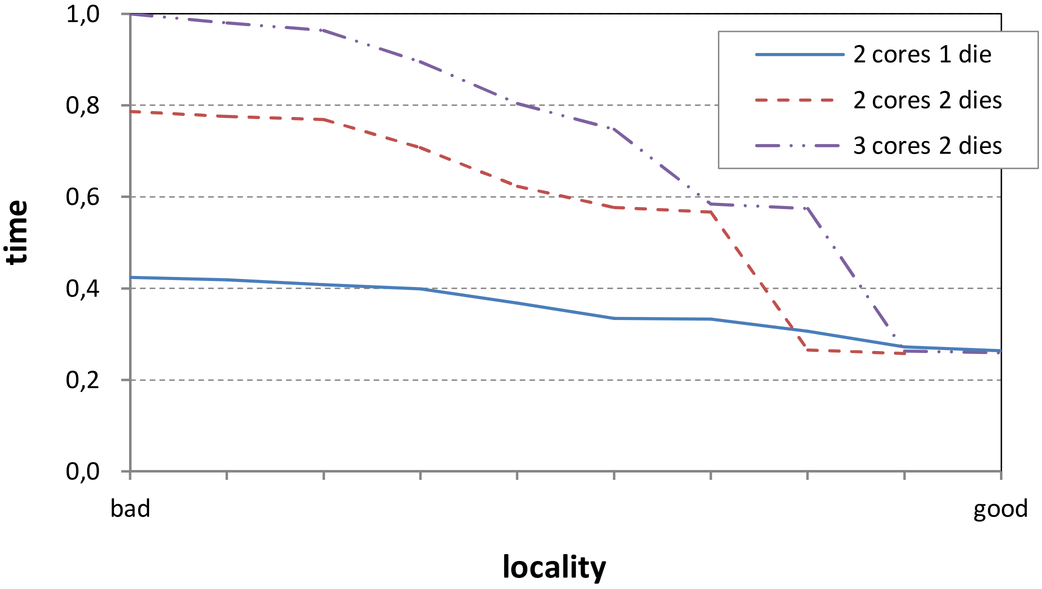
Lokalität	2 Cores 1 *Die*	Speedup	2 Cores 2 *Dies*	Speedup	3 Cores 2 *Dies*	Speedup
1	0,42	1	0,78	1	1	1
2	0,41	1	0,77	1,01	0,98	1,02
7	0,40	1	0,76	1,02	0,96	1,03
8	0,39	1,06	0,70	1,11	0,89	1,11
9	0,36	1,15	0,62	1,26	0,80	1,24
10	0,33	1,27	0,57	1,36	0,74	1,33
15	0,33	1,27	0,56	1,36	0,58	1,71
18	0,30	1,38	0,26	2,96	0,57	1,74
30	0,27	1,55	0,25	2,96	0,26	3,70
156	0,26	16,38	0,25	2,96	0,25	3,80

Tabelle 6.6.: Experiment 1: Power 5 normalisiert auf die längste Ausführungzeit

Experiment 2 Eigentlich wäre damit, analog zur Sun Niagara T1, ein Knick nach der Hardware unterstützenden Anzahl pro Core - im Fall der Power 5+ sind es zwei Threads pro Core - zu erwarten. Da dies nicht der Fall ist, ist anzunehmen, dass ein Core mit der Aufgabe - besonders bei schlechter Datenlokalität - immer noch auf blockierende Threads wartet; bei zweien



(a) unabhängig normalisiert



(b) normalisiert auf die längste Ausführungzeit

Abbildung 6.4.: Experiment 1: Power 5+

hat er keinen den er einschieben könnte.
Bei guter Datenlokalität kann - durch das Einsetzen mehrere Threads pro Core - bei der Verwendung von allen drei zur Verfügung stehenden Cores, nahezu kein Effekt erreicht werden wie aus der durchgezogenen Linie in Abbildung 6.5 (a) zu erkennen ist.

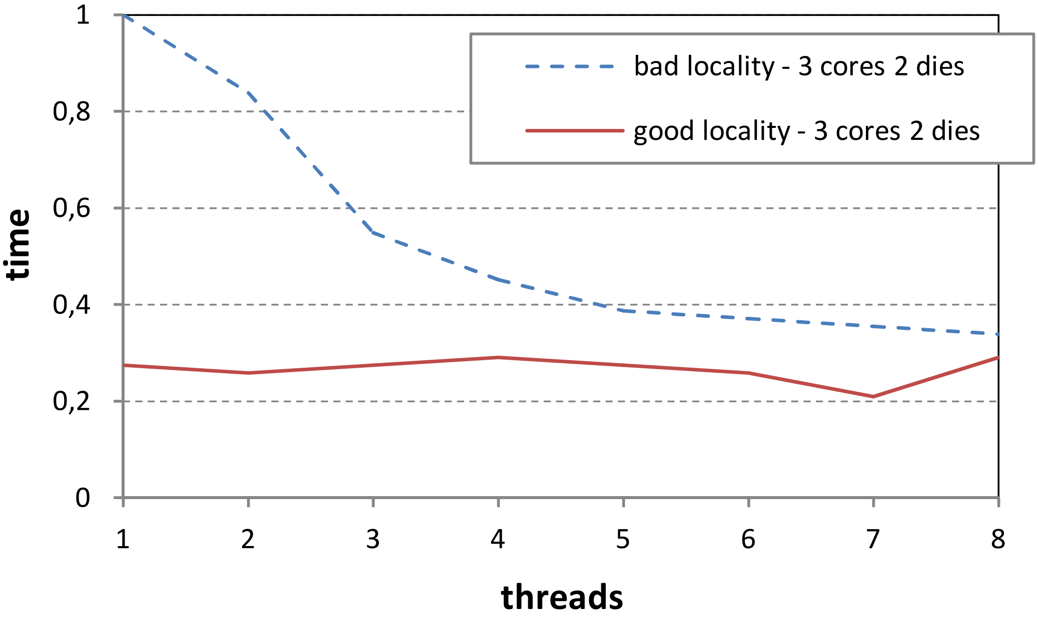
In Abbildung 6.5 (b) ist ersichtlich, dass der Effekt bei zwei Cores auf zwei verschiedenen *Dies* größer ist als wenn die beiden Cores sich auf demselben *Die* befinden. Durch die unterschiedlichen *Dies* blockieren die Threads bei einem Cache Line Transfer länger als wenn sich die Cores, auf denen die Threads liegen, auf demselben *Die* befinden. Daher ist der Speedup bei der Verwendung von zwei Cores auf demselben *Die* auch nur 1,60; bei zwei Cores auf verschiedenen *Dies* hingegen ist der Speedup 2,90. Der Speedup von zwei Cores auf zwei *Dies* - bei der Verwendung von drei Cores auf zwei *Dies* - unterscheidet sich erst in der zweiten Kommastelle.
Die normalisierten Werte der eben erwähnten Grafiken für 3 Cores auf 2 *Dies* befinden sich in Tabelle 6.7, für den Vergleich von 2 Cores auf demselben *Die* und verschiedenen *Dies* in Tabelle 6.8 (die angegebenen Werte sind immer auf die längste Ausführungszeit innerhalb der Tabelle normalisiert) und der dazugehörige Speedup wird extra in Tabelle 6.9 dargestellt.

Threads per Core	Schlechte DL	Speedup	Gute DL	Speedup
1	1	1	0,27	1
2	0,83	1,19	0,25	1,06
3	0,54	1,82	0,27	1
4	0,45	2,21	0,29	0,94
5	0,38	2,58	0,27	1
6	0,37	2,69	0,25	1,06
7	0,35	2,81	0,20	1,30
8	0,33	2,95	0,29	0,94

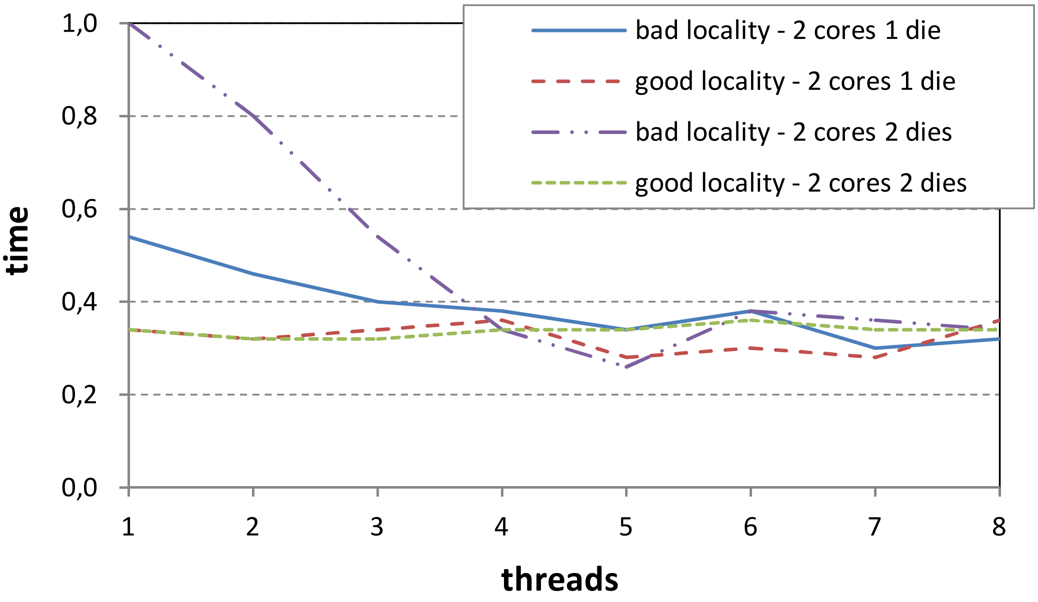
Tabelle 6.7.: Experiment 2: Power 5: 3 Cores auf 2 *Dies*; normalisiert auf die längste Ausführungszeit

6.3. AMD Opteron

Da der Opteron nur zwei Cores (die sich beide auf demselben *Die* befinden) besitzt, gibt es hier nur geringe Variationen in den Berechnungen, beide Experimente werden jeweils mit einem Thread pro Core begonnen.



(a) 3 Cores auf 2 Dies



(b) 2 Cores auf 1 und 2 Dies

Abbildung 6.5.: Experiment 2: Power 5+

Threads per Core	Schlechte DL (2 Cores 2 *Dies*)	Gute DL (2 Cores 2 *Dies*)	Schlechte DL (2 Cores 1 *Die*)	Gute DL (2 Cores 1 *Die*)
1	0,54	0,34	1	0,34
2	0,46	0,32	0,8	0,32
3	0,4	0,34	0,54	0,32
4	0,38	0,36	0,34	0,34
5	0,34	0,28	0,26	0,34
6	0,38	0,3	0,38	0,36
7	0,3	0,28	0,36	0,34
8	0,32	0,36	0,34	0,34

Tabelle 6.8.: Experiment 2: Power 5: 2 Cores: normalisiert auf die längste Ausführungszeit

Threads per Core	Schlechte DL (2 Cores 1 *Die*)	Gute DL (2 Cores 1 *Die*)	Schlechte DL (2 Cores 2 *Dies*)	Gute DL (2 Cores 2 *Dies*)
1	1	1	1	1
2	1,17	1,06	1,25	1,06
3	1,35	1	1,85	1,06
4	1,42	0,94	2,94	1
5	1,58	1,21	,84	1
6	1,42	1,13	2,63	0,94
7	1,80	1,21	2,77	1
8	1,68	0,94	2,94	1

Tabelle 6.9.: Experiment 2: Power 5: 2 Cores: Speedup

Experiment 1 In Abbildung 6.6 ist Experiment 1 unter Verwendung von beiden Cores dargestellt. Diese beiden Cores befinden sich auf demselben *Die*. Dabei wird ein Beschleunigung um den Faktor von 4,2 erreicht. Die Werte in dem Diagramm sind auf die längste Ausführungzeit normalisiert. Im Gegensatz zu den anderen in den Experimenten verwendeten Prozessoren sind bei dem AMD Opteron deutlich Schwankungen zu erkennen, so kann es hier bei besseren Datenlokalität zu einem schlechteren Ergebnis kommen. In Summe betrachtet kann aber auch auf dem AMD Opteron eine deutliche Verkürzung der Ausführungszeit durch Verbessern der Datenlokalität erreicht werden.

In Tabelle 6.10 sind die Werte zu Abbildung 6.6 auf die längste Ausführungszeit normalisiert und der dazugehörige Speedup dargestellt.

Experiment 2 In Abbildung 6.7 befinden sich die Ergebnisse von Experiment 2 auf dem AMD Opteron. Die Ergebnisse sind wieder auf die längste Ausführungzeit normalisiert. Die

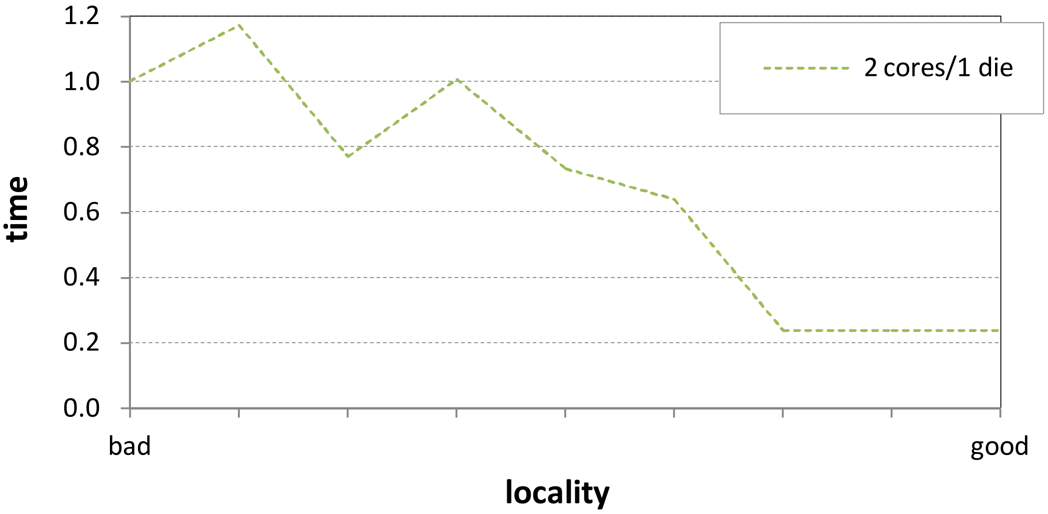


Abbildung 6.6.: Experiment 1: AMD Opteron

Lokalität	2 Cores	Speedup
1	1	1
5	1,17	0,85
8	0,77	1,29
9	1,01	0,99
10	0,73	1,36
14	0,64	1,56
30	0,24	4,18
40	0,24	4,20
156	0,24	4,19

Tabelle 6.10.: Experiment 1: AMD Opteron; normalisiert auf die längste Ausführungszeit

stärkste Verbesserung wird beim Einsatz von zwei Threads auf zwei Cores erreicht. Danach kommt es scheinbar nur mehr zu geringen Absenkungen der Ausführungszeit durch das Aufteilen der Arbeit auf mehrere Threads je Core, wie die durchgezogene Linie in Abbildung 6.7 zeigt. Bei guter Datenlokalität bleibt die Ausführungzeit annähernd gleich, egal wieviele Threads pro Core arbeiten, wie aus der strichlierten Linie in Abbildung 6.7 ersichtlich.

In Tabelle 6.11 sind die auf die längste Ausführungszeit normalisierten Werte und der dazugehörige Speedup für Experiment 2 auf dem AMD Opteron angegeben.

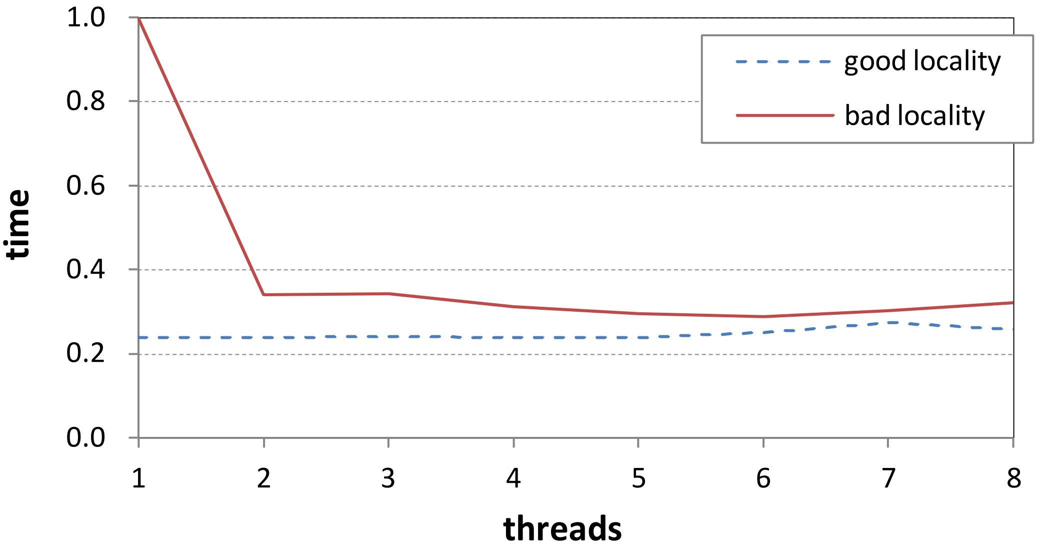


Abbildung 6.7.: Experiment 2: AMD Opteron

Threads per Core	Gute Datenlokalität	Speedup	Schlechte Datenlokalität	Speedup
1	1	1	1	1
2	1	1	2,94	2,93
3	0,99	0,99	2,91	2,91
4	1	1	3,20	3,19
5	4,19	4,19	3,37	3,37
6	3,98	3,97	3,46	3,46
7	3,65	3,64	3,31	3,30
8	3,88	3,88	3,11	3,10

Tabelle 6.11.: Experiment 2: AMD Opteron; normalisiert auf die längste Ausführungszeit

6.4. Intel Core 2 Extreme

Der Intel Core 2 Extreme ist - hinsichtlich der Aufteilung auf die einzelnen Cores - ähnlich dem Power 5, weil es hier ebenfalls immer zwei Cores, die sich auf demselben *Die* befinden, gibt. Bei diesem Experiment stehen allerdings alle vier Cores zur Verfügung. Der Intel Core 2 Extreme unterstützt kein Multithreading, womit ab dem zweiten Thread, der jedem Core zugewiesen wird, zusätzlicher Migrationsaufwand beim Wechsel zwischen den Threads entsteht.

Experiment 1 In Abbildung 6.8 (a) wurden die einzelnen Messungen unabhängig voneinander normalisiert und in (b) auf die längste Ausführungszeit normalisiert. Wie erwartet, ist bei der Verwendung der meisten Cores - bei schlechter Datenlokalität - die Ausführungszeit am größten, weil dabei mehr Threads arbeiten. Bei vier Threads erreicht man einen Speedup von 44,20. Ebenso wie bei der Power 5 ist der Speedup bei zwei Cores auf zwei *Dies* größer als bei zwei Cores auf demselben *Die*. Befinden sich die Cores auf zwei verschiedenen *Dies*, so beträgt er 10,10; wenn sich die beiden Cores auf demselben *Die* befinden, so beträgt er 7,20. In Tabelle 6.12 sind die gemessenen Werte, normalisiert auf die längste Ausführungszeit, und in Tabelle 6.13 wird der dazugehörige Speedup, angegeben.

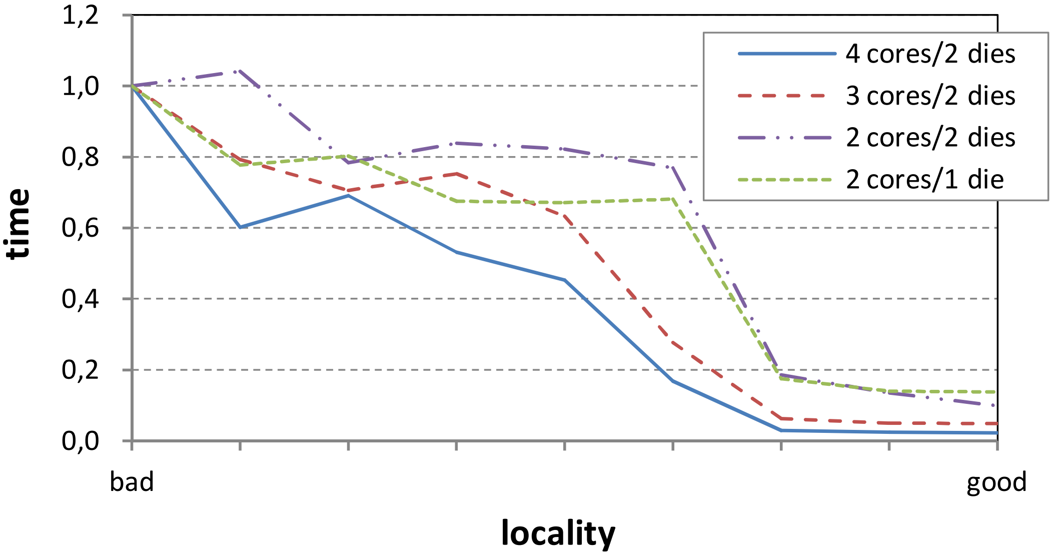
Lokalität	4 Cores	3 Cores	2 Cores 1 *Die*	2 Cores 2 *Dies*
1	1	0,45	0,16	0,22
5	0,60	0,36	0,12	0,23
8	0,69	0,32	0,13	0,17
9	0,53	0,34	0,11	0,19
10	0,45	0,29	0,10	0,18
14	0,16	0,12	0,11	0,17
30	0,02	0,02	0,02	0,04
40	0,02	0,02	0,02	0,03
156	0,022	0,02	0,02	0,02

Tabelle 6.12.: Experiment 1: Intel Core 2 Extreme; normalisiert auf die längste Ausführungszeit

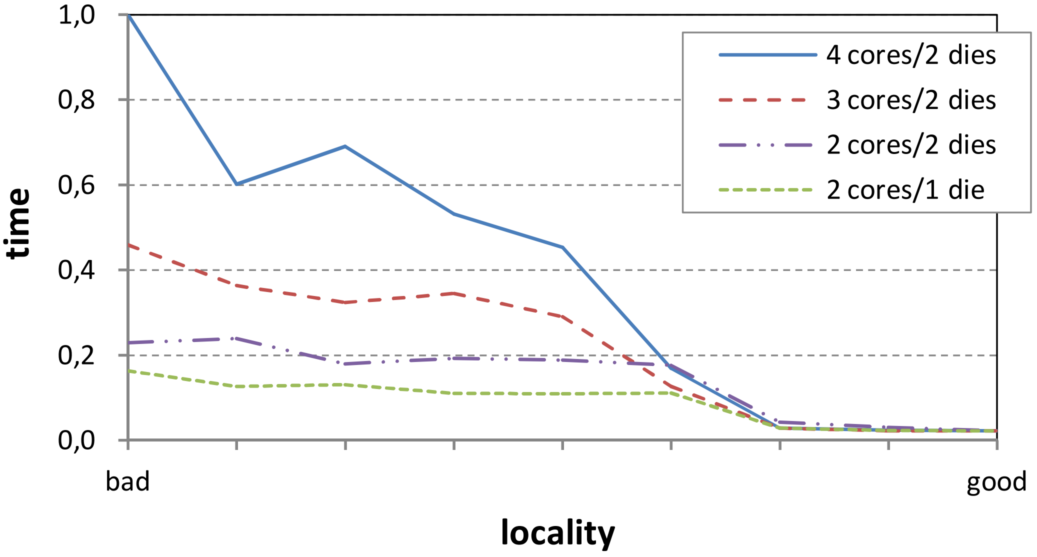
Lokalität	4 Cores	3 Cores	2 Cores 1 *Die*	2 Cores 2 *Dies*
1	1	1	1	1
5	1,66	1,26	1,28	0,96
8	1,44	1,41	1,24	1,27
9	1,88	1,32	1,48	1,19
10	2,20	1,57	1,48	1,21
14	5,93	3,61	1,46	1,30
30	34,70	16,05	5,72	5,39
40	41,42	20,18	7,16	7,43
156	44,25	20,32	7,24	10,16

Tabelle 6.13.: Experiment 1: Intel Core 2 Extreme; Speedup

Experiment 2 Beim Experiment 2 ist vor allem der Unterschied zwischen zwei Cores auf demselben *Die* bzw. auf verschiedenen *Dies* interessant. Ebenso von Bedeutung ist es, wie es bei der Verwendung von allen Cores - in diesem Fall allen vier Cores - aussieht. In Abbildung 6.9 (a) wird gute und schlechte Datenlokalität unter Verwendung von allen vier Cores gegenübergestellt und in 6.9 (b) befindet sich der Vergleich mit zwei Cores auf zwei *Dies* und zwei



(a) unabhängig normalisiert



(b) normalisiert auf die längste Ausführungszeit

Abbildung 6.8.: Experiment 1: Intel Core 2 Extreme

Cores auf demselben *Die*; alle gemessenen Werte wurden auf die längste Ausführungszeit innerhalb eines Diagramms normalisiert.

Auffällig in Abbildung 6.9 (a) und (b) ist, dass bei guter Datenlokalität die Erhöhung der Threads keinen Speeup mehr bringt; der Speedup beträgt bei der Verwendung von allen vier Cores 0,67. Ebenso gibt es sowohl bei zwei Cores auf demselben *Die* als auch auf verschiedenen Dies keinen Ausführungszeitgewinn.

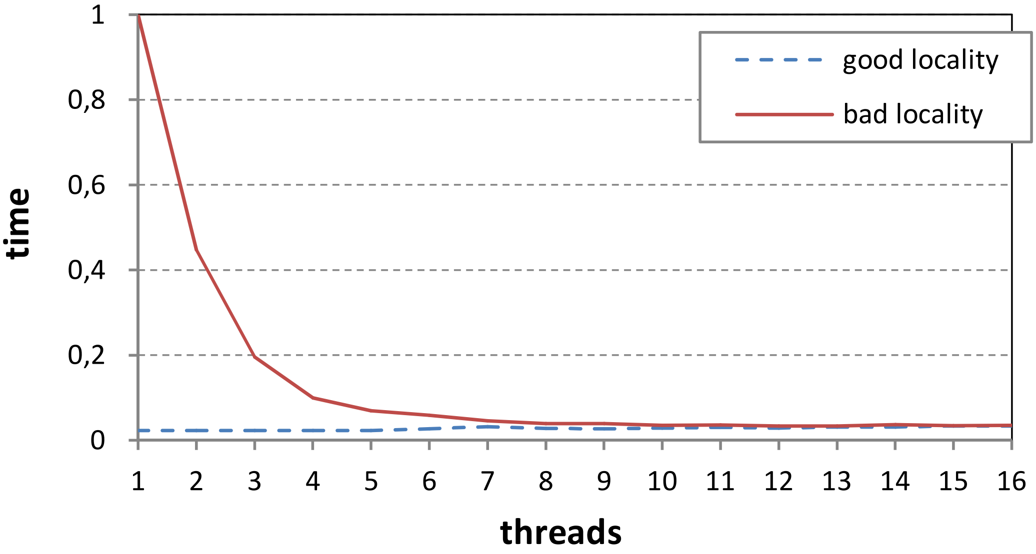
Wie aus Abbildung 6.9 (b) ersichtlich, ist bei schlechter Datenlokalität der Speedup bei zwei Cores auf verschiedenen *Dies* wieder größer als bei zwei Cores auf demselben *Die*; auf verschiedenen *Dies* beträgt er 10,10 und auf demselben *Die* nur 6,40.

In Abbildung 6.9 (a) liegt der Speedup für schlechte Datenlokalität - bei Verwendung von 4 Cores - bei 28,30. Hier werden die Threads im Gegensatz zu den anderen Experimenten aber bis 16 hinaufgezählt.

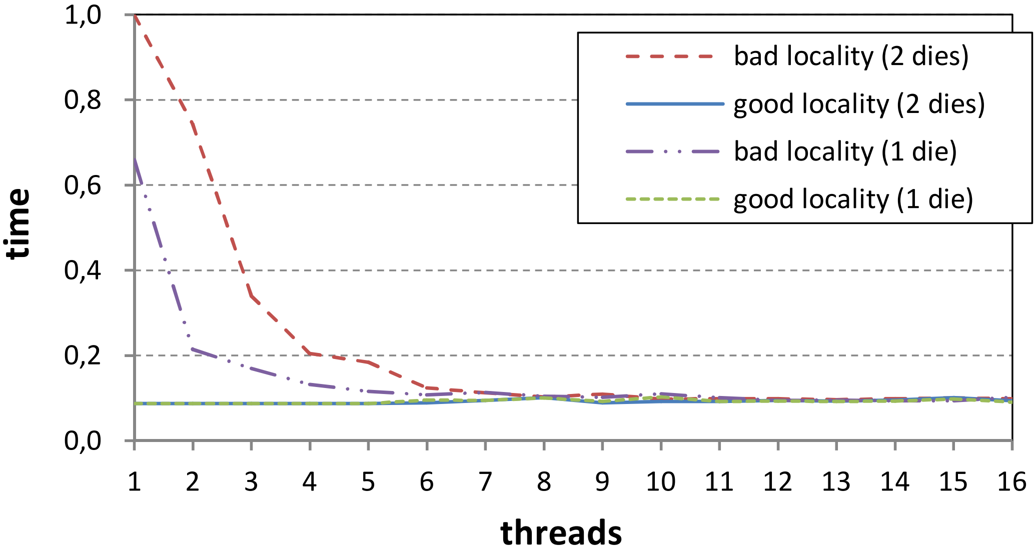
In Tabelle 6.14, 6.15, 6.16 und 6.17 befinden sich die gemessenen Werte für die beiden oben beschriebenen Diagramme, jeweils normalisiert auf die längste Ausführungszeit.

Threads per Core	4 Cores 2 *Dies*	Speedup	3 Cores 2 *Dies*	Speedup
1	1	1	0,45	1
2	2,23	2,23	0,25	1,83
3	5,11	5,11	0,09	4,61
4	10,05	10,05	0,06	6,91
5	14,36	14,36	0,05	8,15
6	16,91	16,91	0,04	11,29
7	21,84	21,84	0,03	12,44
8	25,41	25,41	0,03	13,71

Tabelle 6.14.: Intel Core 2 Extreme Experiment 2: Schlechte Datenlokalität auf 4 und 3 Cores; normalisiert auf die längste Ausführungszeit



(a) 4 Cores



(b) 2 Cores

Abbildung 6.9.: Experiment 2: Intel Core 2 Extreme

Threads per Core	2 Cores 2 *Dies*	Speedup	2 Cores 1 *Die*	Speedup
1	1	1	0,65	1
2	0,74	1,34	0,21	3,07
3	0,33	2,94	0,17	3,88
4	0,20	4,88	0,13	5,00
5	0,18	5,42	0,11	5,70
6	0,12	8,04	0,10	6,14
7	0,11	8,87	0,11	5,81
8	0,10	9,75	0,10	6,28

Tabelle 6.15.: Intel Core 2 Extreme Experiment 2: Schlechte Datenlokalität auf 2 Cores auf einem *Die* und zwei *Dies*; normalisiert auf die längste Ausführungszeit

Threads per Core	4 Cores 2 *Dies*	Speedup	3 Cores 2 *Dies*	Speedup
1	0,02	1	0,02	1
2	0,02	0,99	0,02	0,99
3	0,02	0,99	0,02	0,99
4	0,02	0,98	0,02	0,99
5	0,02	0,83	0,02	0,98
6	0,02	0,71	0,02	0,90
7	0,03	0,81	0,02	0,89
8	0,02	0,84	0,02	0,83

Tabelle 6.16.: Intel Core 2 Extreme Experiment 2: Gute Datenlokalität auf 4 und 3 Cores; normalisiert auf die längste Ausführungszeit

Threads per Core	2 Cores 2 *Dies*	Speedup	2 Cores 1 *Die*	Speedup
1	0,08	1	0,08	1
2	0,08	1	0,08	1
3	0,08	1	0,08	1
4	0,08	1	0,08	1
5	0,08	1	0,08	1
6	0,08	1	0,09	0,91
7	0,09	0,92	0,09	0,91
8	0,10	0,86	0,10	0,86

Tabelle 6.17.: Intel Core 2 Extreme Experiment 2: Gute Datenlokalität auf 2 Cores auf einem *Die* und zwei *Dies*; normalisiert auf die längste Ausführungszeit

7. Diskussion der Ergebnisse

Durch das Zuweisen der Threads auf die einzelnen Cores wird das Migrieren von einzelnen Threads auf andere Cores vollständig unterbunden. Somit wird keine Zeit für das Migrieren von Threads aufgewendet. Wie bereits in Kapitel 6 erwähnt, sind nur die normalisierten Ausführungszeiten zur Betrachtung herangezogen worden. Es wurde also die längste Ausführungszeit 1.0 gesetzt, womit ein Vergleich zwischen den einzelnen Diagrammen vereinfacht wird. Die Messungen in Sekunden befinden sich in Appendix B. Die Experimente dienen allerdings nicht dazu, dass C und Java oder die einzelnen Prozessorarchitekturen miteinander verglichen werden.

7.1. Experiment 1

Eine Übersicht über die maximalen Speedups, die in Experiment 1 erreicht wurden, sind in Tabelle 7.1 dargestellt. Für die Sun T1 wurden die Experimente in C und in Java durchgeführt; für die anderen Hardware Plattformen nur in C. In diesen Abbildungen ist auf der x-Achse der Grad der Datenlokalität eingetragen während auf der y-Achse die normalisierte Ausführungzeit aufscheint. Bei der Betrachtung der Ergebnisse ist zu beachten, dass hier jeweils der/die Thread/s auf einem Core die gleiche Arbeit zu verrichten haben. Wird bei einem Experiment also ein weiterer Core hinzugefügt so kommt es auch in Summe zu mehr Arbeit als wenn der Core nicht dabei wäre.

	Sun T1	IBM Power 5+	Intel Core 2 Extreme	AMD Opteron
all Cores	1,6	3,9	44,2	4,9
2 cores, 2 dies	-	3,1	10,2	-
2 cores, 1 die	-	1,6	7,2	-

Tabelle 7.1.: Experiment 1: Maximale Speedups für gute vs. schlechte Datenlokalität

Das erste Experiment zeigt, dass die Performanceunterschiede zwischen guter und schlechter Datenlokalität von Bedeutung sind. Für die beiden Server Multicores, Sun T1 und IBM Power

5+ erhält man den identischen Speedup von 1,60, wenn sich die verwendeten Cores auf demselben *Die* befinden. Dies ist nur bei der IBM Power 5+ relevant, weil die Sun T1 nur einen *Die* besitzt auf dem alle Cores vorhanden sind. Ähnlich sind die Ergebnisse des AMD Opterons zu betrachten. Hier wurde nur auf einem Opteron gemessen, der aus zwei Cores besteht, die sich auf demselben *Die* befinden, auch hier gibt es keine Vergleichsmessungen zwischen unterschiedlichen *Dies*.
Der Performanceunterschied des AMD Opterons ist allerdings dem anderen Desktop Rechner dem Intel Core 2 Extreme näher als dem der Sun T1. Auf den beiden Desktop Multi-cores - dem Intel Core 2 Extreme und dem AMD Opteron - wurde ein Speedup von 7,10 und 4,90 gemessen, wenn sich die dabei verwendeten Cores auf demselben *Die* befanden. Dieser Unterschied ist im Gegensatz zu den beiden Server Multi-cores sehr groß. Dieses Ergebnis zeigt deutlich, dass man mit guter Datenlokalität bei den Server Multi-cores eine Performance Verbesserung von 60% erreichen kann und dazu, um wievieles wichtiger, gute Datenlokalität bei den Desktop Multi-cores ist. Hier kann der Performanceunterschied schon etwa eine Zehnerpotenz betragen.

Für die Messungen des ersten Experiments waren der IBM Power 5+ und der Intel Core 2 Extreme interessanter, weil hier verschiedene Versuchsanordnungen unter Berücksichtigung der Anordnung der einzelnen Cores auf den *Dies* vorgenommen werden konnten. Hier wurde festgestellt, das ein erheblicher Unterschied in Bezug auf die Latenzzeit bei Cache Misses zwischen Cores die auf den Speicher der sich auf demselben *Die* befindet und dem der sich auf dem anderen *Die* lokalisiert ist.
Durch unterschiedliche Anzahl an Threads und durch Verteilung dieser auf die einzelnen Cores wurde beim IBM Power 5+ folgende Performanceverbesserungen erreicht:

- 1,6 bei zwei Threads auf auf zwei Cores auf einem *Die*,
- 3,1 für zwei Threads auf zwei Cores auf verschiedenen *Dies* und
- 3,9 für drei Threads auf drei Cores die sich auf zwei verschiedenen *Dies* befanden.

Dies ist aus den den Abbildungen 6.4 (a) und (b) zu entnehmen.

Für den Intel Core 2 Extreme wurde ein Speedup bis zu 7,2 für zwei Threads auf zwei Cores, die sich auf demselben *Die* befinden und ein Speedup von 10,2 bei zwei Threads auf zwei Cores auf unterschiedlichen *Dies*, erreicht. Bei drei Threads auf drei Cores, die sich auf zwei *Dies* befanden wurde ein Speedup von 20,3 erreicht und bei vier Threads auf vier Cores auf zwei *Dies* konnte ein Speedup 44,2 - wie aus dem Abbildungen 6.8 (a) und (b) zu entnehmen ist - erzielt werden. Hier ist ebenso wie beim IBM Power 5+ der Zugriff auf gemeinsame Daten effizienter, wenn sich die Cores auf demselben *Die* befinden.

In den Abbildungen 6.4 und 6.8 sind alle verwendeten Prozessoren bei Experiment 1 auf die höchste Ausführungzeit

- normalisiert in (b)
- und unabhängig voneinander normalisiert in (a) dargestellt.

Wie erwartet, wurde die Ausführungzeit mit dem Hinzufügen jedes Cores bei schlechter Datenlokalität langsamer. Dabei ist besonders zu beachten, dass auch mit dem Hinzufügen jedes Cores ein Teil Arbeit hinzukommt, weil jeder Core den gleichen Anteil an Tätigkeit zu verrichten hat. Die Diagramme zeigen für alle Prozessoren dasselbe Verhalten.
Ähnlich ist es auch bei der Sun T1 (wie in Abbildung 6.2 dargestellt). Hier wird - jeweils normalisiert auf die höchste Ausführungszeit - der Performanceunterschied nicht so groß wie bei den Prozessoren, wo zwischen verschiedenen *Dies* unterschieden werden kann. Man erhält eine Faktor von 1,2, wenn man von vier auf zwei Cores heruntergeht und einen Faktor von 1,3, wenn man von sechs auf zwei Cores absenkt.
Auffällig bei diesen Messungen ist auch, dass die Anzahl der Cores bei guter Datenlokalität keine signifikante Rolle spielt. Einzige bei der Sun T1 gibt es - wie bereits erwähnt - Performanceunterschiede. Dies lässt sich vermutlich darauf zurückführen, dass sich hier alle Cores auf demselben *Die* befinden. Bei schlechter Datenlokalität hingegen ist der Performanceunterschied geringer, wenn sich die Cores auf demselben *Die* befinden. Dies ist nicht besonders überraschend, weil jeder *Die* einen eigenen L2 Cache besitzt und bei schlechter Datenlokalität auf zwei verschiedenen *Dies* immer auf den L2 Cache eines anderen *Dies* zugegriffen werden muss.

7.2. Experiment 2

Im zweiten Experiment wurden die Berechnungskernels für gute und schlechte Datenlokalität verwendet, wobei versucht wurde die *Stalls*, die durch die Cache Misses entstanden sind, durch die höhere Anzahl der Threads pro Cores zu verstecken.

Eine Übersicht über die maximal erreichten Speedups in Experiment 2 sind in Tabelle 7.2 zu finden.

	Sun T1	IBM Power 5+	Intel Core 2 Extreme	AMD Opteron
schlechte Datenlokalität				
all Cores	2,8	3,1	30,1	3,5
2 cores, 2 dies	-	2,9	10,4	-
2 cores, 1 die	-	1,8	7,0	-
gute Datenlokalität				
all Cores	1,9	0,9	0,7	0,9
2 cores, 2 dies	-	0,9	0,9	-
2 cores, 1 die	-	0,9	0,9	-

Tabelle 7.2.: Experiment 2: Maximale Speedups für gute vs. schlechte Datenlokalität

Die Diagramme zeigen, dass bei schlechte Datenlokalität immer ein deutlicher Speedup durch das Hinzufügen von zusätzlichen Threads zu einem Core erhalten wird. Für die Server Multi-Cores liegt der Speedup etwa bei 3 während für die Desktop Multi-Cores der Speedup zwischen 7 und 30 liegt. Dies zeigt, dass das Erzeugen von mehreren Threads eine gute Strategie ist um einen Performancegewinn zu erzielen. Bei guter Datenlokalität kann es zu kleinen Performanceverlusten beim Hinzufügen von zu vielen Threads kommen, wie es bei dem IBM Power 5+ und dem Intel Core 2 Extreme auffällt. Dies kann vermutlich durch den Overhead, der durch das Wechseln der Threads entsteht, erklärt werden. Es scheint so, als sei dieser Overhead größer als die Performanceverbesserung, die durch eine bessere Auslastung der Prozessoren erreicht werden kann. Nur für den Sun T1 kann auch bei guter Datenlokalität durch das Erzeugen von mehreren Threads ein weiterer Performancegewinn erzielt werden. Der Sun T1 hat nur eine eingeschränkte Unterstützung für ILP und kann nur eine Instruktion per Clock Cycle ausführen. Dadurch ist der TLP bei der Sun T1 von besonderer Bedeutung, weil hier nur so die Prozessoren ausgelastet werden können.

Die Ergebnisse bei der Sun T1 haben klar auf den Hardwaresupport von 4 *Strands* pro Core hingewiesen. Wird die Anzahl der Threads auf einem Core auf fünf heraufgesetzt, so ergibt sich klar ein Performanceverlust gegenüber der Verwendung von vier Threads pro Core. Auch beim IBM Power 5+ ist ein leichte Rückgang der Performanceverbesserung zu sehen, wenn von zwei auf drei Threads erhöht wird, weil der IBM Power 5+ eine Hardwareunterstützung von zwei Threads pro Core bietet.

In allen Experimenten ist der größte Performancegewinn beim Erhöhen auf zwei Threads pro Core erreicht worden. Normalerweise haben auch noch drei Threads per Core positive Auswirkungen, wobei allerdings beim Überschreiten der Anzahl von vier Threads pro Core keine Performancegewinne mehr eingetreten sind.

Der Unterschied zwischen guter und schlechter Datenlokalität ist - durch das Hinzufügen von weiteren Threads per Core - in allen Experimenten geringer geworden.

8. Zusammenfassung

In der Arbeit wurde auf die Auswirkungen, welche durch die Entwicklung von multi-core Architekturen verursacht wurden für die Softwareentwicklung, besonders im Bezug auf Anwendungen aus Computational Science betrachtet. Ein besonderes Augenmerk wurde dabei auf das Prinzip der Datenlokalität gelegt. Einerseits sollte durch Experimente die Bedeutung von guter Datenlokalität dargestellt werden. Da aber gute Datenlokalität nicht immer einfach zu erreichen ist wurde auch nach einer Strategie gesucht wie man trotz schlechter Datenlokalität auf multi-core Architekturen einen Performancegewinn erzielen kann. Bei diesen synthetischen Experimenten wurde jeweils die Ausführungszeit gemessen.

Durch Experiment 1 wurde eindeutig die Auswirkungen von Datenlokalität für Scientific Anwendungen bei der Verwendung von multi-core Architekturen dargestellt. Für die beiden in den Experimenten verwendeten Server Architekturen sind die Auswirkungen nicht so groß ausgefallen wie vielleicht anfangs erwartet. Bei den beiden Desktop Architekturen hingegen wurden erheblich Unterschiede in der Ausführungzeit - bei guter und schlechter Datenlokalität - gemessen. Als einfach Art der Optimierung hat sich das Platzieren von Threads, die auf dem gleichen Bereich des Arrays arbeiten und auf Cores, die sich auf demselben *Die* befinden erwiesen.

Weiteres wurde durch Experiment 2 aufgezeigt, dass sich die Aufteilung einer Arbeit auf mehrere Threads positiv auf die Performance auswirkt. Dies besonders, wenn diese Threads auf denselben Core zugewiesen werden, um dort die Prozessore Stalls, die aus den Cache Misses resultieren, auszugleichen. Damit wird der Prozessor soweit als nur möglich ausgelastet, wenn die Arbeit auf ausreichend Threads aufgeteilt wurde.

Die beiden Experimente wurden auf zwei Server- und zwei Desktop Architekturen durchgeführt. Für die Messungen auf den Serverarchitekturen diente von Sun der UltraSPARC T1 und von IBM der Power 5+. Die beiden Desktoprechner waren von Intel der Core 2 Extreme und von AMD der Opteron.

9. Literaturverzeichnis

[1] AMD. Amd opteron homepage. Internet, 2007. http://www.amd.com/opteron.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, 33(4):80–91, 2005.

[5] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1):3–10, February 2007.

[6] John L. Hennessy and David A. Patterson. *Computer Organisation Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[8] IBM. *IBM AIX Manual*, 2007.

[9] IBM. Ibm system i - dynamic logical partitioning. ibm.com, 2007. http://www-03.ibm.com/servers/eserver/iseries/lpar/.

[10] Intel. Intel core2extreme processor - technical specifications. intel.com, 2007. http://www.intel.com/products/processor/core2XE/index.htm.

[11] Sheng Liang. *The Java Native Interface: Programming Guide and Reference*. Addison-Wesley Longman, 1999.

[12] Sun Microsystems. Sun homepage - ultrasparc t2 processor - technical specifications. Internet, 2007. http://www.sun.com/processors/UltraSPARC-T2/specs.xml.

[13] Christian Märtin. *Einführung in die Rechnerarchitektur*. Fachbuchverlag Leipzig, 2003.

[14] Linux Man Pages. *Linux Manual*, 2007.

[15] Denis Sheahan. Developing and tuning applications on ultrasparc t1 chip multithreading systems. January 2007. http://docs.sun.com/blueprints.

[16] B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickemeyer, and J.B. Joyner. Power5 system microarchitekture. IBM J. RES DEV, Juli/September 2005.

[17] Thomas Sterling, Peter Kogge, William J Dally, Steve Scott, William Gropp, David Keyes, and Pete Beckman. Multi-core for hpc: breakthrough or breakdown? In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 73, New York, NY, USA, 2006. ACM.

[18] Sun. *Sun Solaris Manual*, 2006.

[19] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, 2007.

A. Java Source für Sun T1

In Listing A.1 wird der Source Code abgebildet, wie er auf der Sun T1 für die Experimente verwendet wurde. Dieser Code ist soweit parametrisiert, dass er für beide Experimente verwendet werden kann.

Beim Aufruf der Main Methode werden die benötigten Parameter für die LINESZ erhalten. Diese soll den Grad der Parallelität (in den Messungen befand sich der Zahlenbereich zwischen 1 und 156) darstellen. Die Anzahl der Threads je Core, den Index des ersten Cores, der verwendet werden soll (diese beginnen bei 0 zu zählen) und die Anzahl der verwendeten Cores werden als Parameter übergeben.

```java
/*
 * MultiCoreTest.java
 *
 * Created on 1. July 2007, 02:06
 *
 */

package at.ac.univie.isc.multicoretest;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

/**
 *
 * @author mabauer
 */
public class MultiCoreTest extends Thread{

    private static CyclicBarrier BARRIER;

    final private static int STRANDS = 4; //Hardware Characteristic of T1

    private static int TOTAL_NUM_THREADS;
    private static int LINESZ;
    private static int FIRST_CORE;
    private static int CORES_USED;
```

```
    private static int THREADS_PER_CORE;

    private static int[] ARR = new int[10000];

    private int mCore;
    private int mThreadOnCore;

    /** Creates a new instance of MultiCoreTest */
    public MultiCoreTest(int core, int threadOnCore) {
        super();
        Properties prop = new Properties();
        try{
            prop.load(new FileInputStream("config.properties"));
            System.load(prop.getProperty("native-path"));
        } catch(IOException e){
            System.out.println("No Native Library Found! Check config.propöerties File!");
            e.printStackTrace();
        }

    }

    public native void bind(int cpu);

    public void run(){

        int index = 0;

        int cpu = (mCore*STRANDS)+(mThreadOnCore%STRANDS);
        this.bind(cpu);
        index = (cpu/STRANDS)*(LINESZ-1) + (mThreadOnCore%STRANDS)*31;

        System.out.println("Creating thread on CPU " +cpu +" core " +mCore +" id " +
             mThreadOnCore +" index: " +index);

        try {
            BARRIER.await();
        } catch (BrokenBarrierException ex) {
            ex.printStackTrace();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }

        long time = System.nanoTime();

        for(int i = 0; i < (10000*10000)/THREADS_PER_CORE; i++){
            for(int j = 0; j < 10; j++){
                ARR[index+j]=ARR[index+j+1];
            }
        }

        time = System.nanoTime() - time;
        System.out.println(index +" cpu: " +cpu +" elapsed time: " +time);

```

```
    }

    public static void main(String[] args) throws IOException, InterruptedException{
        try{
            LINESZ = Integer.parseInt(args[0]);
            THREADS_PER_CORE = Integer.parseInt(args[1]);
            FIRST_CORE = Integer.parseInt(args[2]);
            CORES_USED = Integer.parseInt(args[3]);

            TOTAL_NUM_THREADS = CORES_USED*THREADS_PER_CORE;

            BARRIER = new CyclicBarrier(TOTAL_NUM_THREADS);

            int core, thread;

            for(int i = 0; i < ARR.length; i++){
                ARR[i]=1;
            }

            MultiCoreTest[] threads = new MultiCoreTest[TOTAL_NUM_THREADS+1];

            int t=0;
            for(core=FIRST_CORE; core < FIRST_CORE+CORES_USED; core++){
                for(thread=0; thread<THREADS_PER_CORE; thread++){
                    threads[t] = new MultiCoreTest(core, thread);
                    threads[t].start();
                    t++;
                }
            }

            for(int i = 0; i < TOTAL_NUM_THREADS; i++){
                threads[i].join();
            }
        } catch(ArrayIndexOutOfBoundsException e){
            System.out.println("usage: at.ac.univie.isc.multicoretest,MultiCoreTest <linesz> <threads_per_core> <first_core> <cores_used>");
        }
    }

}
```

Listing A.1: MultiCoreTest.java

In Listing A.2 wird das von *javah* Generator erzeugte File dargestellt, welches zur Implementierung der nativen Methode in C benötigt wird.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class at_ac_univie_isc_multicoretest_MultiCoreTest */

#ifndef _Included_at_ac_univie_isc_multicoretest_MultiCoreTest
#define _Included_at_ac_univie_isc_multicoretest_MultiCoreTest
#ifdef __cplusplus
extern "C" {
#endif
#undef at_ac_univie_isc_multicoretest_MultiCoreTest_MIN_PRIORITY
#define at_ac_univie_isc_multicoretest_MultiCoreTest_MIN_PRIORITY 1L
#undef at_ac_univie_isc_multicoretest_MultiCoreTest_NORM_PRIORITY
#define at_ac_univie_isc_multicoretest_MultiCoreTest_NORM_PRIORITY 5L
#undef at_ac_univie_isc_multicoretest_MultiCoreTest_MAX_PRIORITY
#define at_ac_univie_isc_multicoretest_MultiCoreTest_MAX_PRIORITY 10L
#undef at_ac_univie_isc_multicoretest_MultiCoreTest_STRANDS
#define at_ac_univie_isc_multicoretest_MultiCoreTest_STRANDS 4L
/*
 * Class:     at_ac_univie_isc_multicoretest_MultiCoreTest
 * Method:    bind
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_at_ac_univie_isc_multicoretest_MultiCoreTest_bind
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
\caption{at\_ac\_univie\_isc\_multicoretest\_MultiCoreTest.h}
```

Listing A.2: at_ac_univie_isc_multicoretest_MultiCoreTest.h

In Listing A.3 ist die Implementierung der naiven Methode in C für die Sparc Plattform dargestellt.

```
#include <pthread.h>
#include <stdio.h>

#include <unistd.h>
#include <sys/processor.h>
#include <sys/procset.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/procfs.h>
#include <sys/lwp.h>

#include "at_ac_univie_isc_multicoretest_MultiCoreTest.h"

JNIEXPORT void JNICALL Java_at_ac_univie_isc_multicoretest_MultiCoreTest_bind
  (JNIEnv *, jobject, jint);
{
        int cpuc = (int)cpu;
        int rc;
        rc = processor_bind(P_LWPID, P_MYID, cpuc, NULL);
}


\caption{cpuBind.c}
```

Listing A.3: cpuBind.c

B. Gemessene Werte der Experimente

Die Zahlen im nachfolgenden Anhang sollen nicht dazu dienen die einzelnen Prozessoren miteinander zu vergleichen. Ebenso können aus diesen Zahlen keine Schlüsse in Bezug auf Performanceunterschiede in C und Java gezogen werden.
Die in der Diplomarbeit erläuterten Experimente sind nicht dazu geeignet Performance-Charakterisitiken von einzelnen Architekturen zu vergleichen.

B.1. Experiment 1

B.1.1. Sun UltraSPARC T1 Niagara

In Tabelle B.1 befinden sich die in C gemessenen Werte für 6 Threads, die auf 6 Cores verteilt waren sowie der jeweils dazugehörige Speedup. In Tabelle B.2 befinden sich alle gemessenen Zeiten für Experiment 1 in Java. Hier wurden 6 Threads auf 6 Cores, 4 Threads auf 4 Cores und 2 Threads auf 2 Cores gemessen; hinter den jeweiligen Zeitangaben ist auch immer der Speedup zu finden.

Lokalität	6 Cores	Speedup
1	28,23	1
2	27,80	1,01
3	26,25	1,07
4	24,32	1,16
5	23,03	1,23
6	21,26	1,33
10	19,07	1,47
12	17,53	1,61
40	17,59	1,60
156	17,61	1,60

Tabelle B.1.: Sun T1 Experiment 1-C

Lokalität	6 Cores	Speedup	4 Cores	Speedup	2 Cores	Speedup
1	73,42	1	67,48	1	56,46	1
2	66,96	1,09	65,32	1,03	55,98	1,01
3	61,91	1,18	61,90	1,09	55,82	1,01
4	58,36	1,25	59,49	1,13	55,92	1,01
5	55,82	1,31	57,99	1,16	55,80	1,01
6	56,39	1,30	57,77	1,16	55,59	1,02
10	53,89	1,36	56,71	1,18	51,83	1,08
12	51,89	1,41	53,30	1,26	50,95	1,10
40	49,61	1,47	51,16	1,31	50,95	1,10
156	43,30	1,52	51,35	1,31	50,95	1,10

Tabelle B.2.: Sun T1 Experiment 1-Java

B.1.2. IBM Power 5+

In Tabelle B.3 befinden sich die in Sekunden gemessenen Werte für Experiment 1 mit 3 Cores auf zwei verschiedenen *Dies* und dem dazugehörigen Speedup. In Tabelle B.4 befinden sich die Messwerte bei der Verwendung von 2 Cores; einmal auf demselben *Die* und einmal auf verschiedenen *Dies*.

Lokalität	3 Cores 2 *Dies*	Speedup
1	64	1
2	62	1,02
7	62	1,03
8	57	1,11
9	51	1,24
10	48	1,33
15	37	1,71
18	36	1,74
30	17	3,7
156	17	3,8

Tabelle B.3.: IBM Power 5+ Experiment 1: 3 Cores auf 2 *Dies*-C

B.1.3. AMD Opteron

In Tabellen B.5 befinden sich die in Sekunden gemessenen Werte für den AMD Opteron für Experiment 1.

Lokalität	2 Cores 2 *Dies*	Speedup	2 Cores 1 *Die*	Speedup
1	50	1	27	1
2	49	1,01	27	1
7	49	1,02	27	1
8	45	1,11	25	1,06
9	39	1,26	24	1,15
10	37	1,36	21	1,27
15	37	1,36	22	1,27
18	17	2,96	19	1,38
30	17	2,96	18	1,55
156	17	2,96	17	16,83

Tabelle B.4.: IBM Power 5+ Experiment 1: 2 Cores auf 2 *Dies* und 2 Cores auf 1 *Die*-C

Lokalität	2 Cores	Speedup
1	24,36	1
5	28,57	0,85
8	18,75	1,29
9	24,55	0,99
10	17,87	1,36
14	15,54	1,56
30	5,82	4,18
40	5,78	4,20
156	5,80	4,19

Tabelle B.5.: AMD Opteron Experiment 1

B.1.4. Intel Core 2 Extreme

In Tabelle B.6 sind die Messergebnisse in Sekunden für 4 Threads auf 4 Cores und 3 Threads auf 3 Cores und der dazugehörige Speedup dargestellt. In Tabelle B.7 findet man die Messergebnisse in Sekunden für 2 Cores; einmal auf demselben *Die* und einmal auf verschiedenen *Dies*.

Lokalität	4 Cores 2 *Dies*	Speedup	3 Cores 2 *Die*	Speedup
1	167,64	1	76,90	1
5	100,94	1,66	60,96	1,26
8	115,75	1,44	54,23	1,41
9	89,09	1,88	57,89	1,32
10	75,91	2,20	48,71	1,57
14	28,27	5,93	21,29	3,61
30	4,83	34,70	4,79	16,05
40	4,04	41,42	3,81	20,18
156	3,78	44,25	3,78	20,32

Tabelle B.6.: Intel Core 2 Extreme Experiment 1: 4 Cores auf 2 *Dies* und 3 Cores auf 2 *Dies*-C

Lokalität	2 Cores 2 *Dies*	Speedup	2 Cores 1 *Die*	Speedup
1	38,46	1	27,41	1
5	40,05	0,96	21,28	1,28
8	30,13	1,27	21,98	1,24
9	32,25	1,19	18,50	1,48
10	31,61	1,21	18,40	1,48
14	29,56	1,30	18,66	1,46
30	7,13	5,39	4,78	5,72
40	5,17	7,43	3,82	7,16
156	3,78	10,16	3,78	7,24

Tabelle B.7.: Intel Core 2 Extreme Experiment 1: 2 Cores auf 2 *Dies* und 2 Cores auf 1 *Die*-C

B.2. Experiment 2

B.2.1. Sun UltraSPARC T1 Niagara

Tabelle B.8 zeigt die Ergebnisse der Messungen für Experiment 2 für gute und schlechte Datenlokalität in Java, die Zeitangaben sind jeweils in Sekunden.

In Tabelle B.9 sind die Ergebnisse der Messungen in C für gute und schlechte Datenlokalität und jeweils der dazugehörige Speedup angeführt.

B.2.2. IBM Power 5+

In Tabelle B.10 findet man die Ergebnisse der Messungen für Experiment 2 für gute und schlechte Datenlokalität mit 3 Cores auf 2 *Dies* mit dem dazugehörigen Speedup; Messergeb-

Threads per Core	Schlechte DL	Speedup	Gute DL	Speedup
1	73,46	1	48,39	1
2	41,05	1,78	27,40	1,76
3	29,04	2,51	21,71	2,22
4	24,89	2,93	27,36	1,76
5	48,46	1,50	38,22	1,26
6	48,52	1,50	39,13	1,23
7	38,72	1,88	32,36	1,49
8	34,18	2,13	29,53	1,63

Tabelle B.8.: Sun T1 Experiment 2: Java

Threads per Core	Schlechte DL	Speedup	Gute DL	Speedup
1	28,23	1	17,61	1
2	15,75	1,79	9,99	1,76
3	12,17	2,31	9,19	1,91
4	11,19	2,52	9,21	1,91
5	13,10	2,15	10,85	1,62
6	11,50	2,45	10,85	1,62
7	10,44	2,70	9,56	1,84
8	10,07	2,80	9,60	1,83

Tabelle B.9.: Sun T1 Experiment 2: C

nisse sind in Sekunden zu verstehen.

In Tabelle B.11 und B.12 sind die Ergebnisse von jeweils zwei Cores auf einem und zwei Cores mit dem jeweils dazugehörigen Speedup zu finden.

B.2.3. AMD Opteron

In Tabelle B.13 sieht man die auf dem AMD Opteron gemessenen Werte für Experiment 2 in Sekunden.

B.2.4. Intel Core 2 Extreme

Tabelle B.14 zeigt die Ergebnisse in Sekunden und den dazugehörigen Speedup für das Experiment 2 mit schlechter Datenlokalität auf drei und vier Cores. In Tabelle B.15 sind die

Threads per Core	Schlechte DL	Speedup	Gute DL	Speedup
1	62	1	17	1
2	52	1,19	16	1,06
3	34	1,82	17	1
4	28	2,21	18	0,94
5	24	2,58	17	1
6	23	2,69	16	1,06
7	22	2,81	13	1,30
8	21	2,95	18	0,94

Tabelle B.10.: IBM Power 5+ Experiment 2: 3 Cores auf 2 *Dies*

Threads per Core	2 Cores 2 *Dies* Schlechte DL	Speedup	2 Cores 2 *Dies* Gute DL	Speedup
1	50	1	17	1
2	40	1,25	16	1,06
3	27	1,85	16	1,06
4	17	2,94	17	1
5	13	3,84	17	1
6	19	2,63	18	0,94
7	18	2,77	17	1
8	17	2,94	17	1

Tabelle B.11.: IBM Power 5+ Experiment 2: 2 Cores auf 2 *Dies*

Ergebnisse für zwei Cores (einmal auf demselben *Die* und einmal auf zwei verschiedenen *Dies*) zu finden.

In Tabelle B.16 befinden sich die Ergebnisse in Sekunden mit dem dazugehörigen Speedup für das Experiment 2 (mit guter Datenlokalität auf drei und vier Cores). Tabelle B.17 stellt die Ergebnisse für zwei Cores (einmal auf demselben *Die* und einmal auf zwei verschiedenen *Dies*) dar.

Threads per Core	2 Cores 1 *Die* Schlechte DL	Speedup	2 Cores 1 *Die* Gute DL	Speedup
1	27	1	17	1
2	23	1,17	16	1,06
3	20	1,35	17	1
4	19	1,42	18	0,94
5	17	1,58	14	1,21
6	19	1,42	15	1,13
7	15	1,80	14	1,21
8	16	1,68	18	0,94

Tabelle B.12.: IBM Power 5+ Experiment 2: 2 Cores auf 1 *Die*

Threads per Core	Schlechte DL	Speedup	Gute DL	Speedup
1	24,36	1	5,80	1
2	8,29	2,93	5,82	0,99
3	8,36	2,91	5,84	0,99
4	7,62	3,19	5,82	0,99
5	7,22	3,37	5,80	4,19
6	7,04	3,46	6,12	3,97
7	7,37	3,30	6,67	3,64
8	7,83	3,10	6,27	3,88

Tabelle B.13.: AMD Opteron Experiment 2

Threads per Core	4 Cores 2 *Dies*	Speedup	3 Cores 2 *Dies*	Speedup
1	167,64	1	75,35	1
2	75,06	2,23	41,13	1,83
3	32,79	5,11	16,32	4,61
4	16,68	10,05	10,89	6,91
5	11,67	14,36	9,24	8,15
6	9,91	16,91	6,67	11,29
7	7,67	21,84	6,05	12,44
8	6,59	25,41	5,49	13,71

Tabelle B.14.: Intel Core 2 Extreme Experiment 2: Schlechte Datenlokalität auf 4 und 3 Cores

Threads per Core	2 Cores 2 *Dies*	Speedup	2 Cores 1 *Die*	Speedup
1	43,39	1	28,62	1
2	32,24	1,34	9,30	3,07
3	14,74	2,94	7,37	3,88
4	8,75	4,88	5,71	5,00
5	8,00	5,42	5,02	5,70
6	5,39	8,04	4,65	6,14
7	4,89	8,87	4,90	5,81
8	4,47	9,75	4,55	6,28

Tabelle B.15.: Intel Core 2 Extreme Experiment 2: Schlechte Datenlokalität auf 2 Cores auf einem *Die* und zwei *Dies*

Threads per Core	4 Cores 2 *Dies*	Speedup	3 Cores 2 *Dies*	Speedup
1	3,78	1	3,78	1
2	3,81	0,99	3,79	0,99
3	3,81	0,99	3,79	0,99
4	3,84	0,98	3,80	0,99
5	3,85	0,83	3,82	0,98
6	4,54	0,71	4,19	0,90
7	5,03	0,81	5,21	0,89
8	4,61	0,84	4,54	0,83

Tabelle B.16.: Intel Core 2 Extreme Experiment 2: Gute Datenlokalität auf 4 und 3 Cores

Threads per Core	2 Cores 2 *Dies*	Speedup	2 Cores 1 *Die*	Speedup
1	3,78	1	3,78	1
2	3,78	1	3,78	1
3	3,78	1	3,78	1
4	3,78	1	3,78	1
5	3,78	1	3,78	1
6	3,87	0,97	4,14	0,91
7	4,10	0,92	4,12	0,91
8	4,38	0,86	4,36	0,86

Tabelle B.17.: Intel Core 2 Extreme Experiment 2: Gute Datenlokalität auf 2 Cores auf einem *Die* und zwei *Dies*