FHZ → FACHHOCHSCHULE ZENTRALSCHWEIZ

HTA → HOCHSCHULE FÜR TECHNIK+ARCHITEKTUR LUZERN

Abteilung Informatik

Java

Thread Skalierung

Conclusion

Schlussfolgerungen Projektergebnisse

HTA Horw

HTA LUZERN
Technikumstrasse 21
CH—6048 Horw

T: 041-349-33-11
F: 041-349-39-60
W: www.hta.fhz.ch

Diplomarbeit 2006
Aregger Marcel
Meier Rainer

Änderungskontrolle

Version	Datum	Ausführende Stelle	Bemerkungen/Art der Änderung
1.1	2006-10-16	Rainer Meier	Initial Release

Prüfung und Freigabe

Vorname/Name	Dokumentversion	Status	Datum	Visum
Rainer Meier	1.2	Draft	2006-11-10	RME
Marcel Aregger	1.2	Draft	2006-11-10	AREGMA

1. Management Summary

Dieses Dokument beinhaltet eine umfassende Analyse der in [2] erhaltenen Messresultate. Das Dokument ist dabei in die folgenden Bereiche gegliedert:

- Testcases Prioritäten (siehe Kapitel 5)
- Testcases Affinität (siehe Kapitel 6)
- Testcases Skalierung (siehe Kapitel 7)

Im Bereich der Prioritäten (siehe Kapitel 5) wurde die Abbildung von Java Thread-Prioritäten auf Windows XP Basisprioritäten ebenso untersucht wie deren Auswirkung auf das Scheduling-Verhalten. Wir konnten feststellen, dass die Abbildung linear erfolgt. Ausserdem überlappen sich die Bereiche zweier benachbarter Prozessprioritätsklassen. Weiter konnten wir beweisen, dass alleine die aus Prozessprioritätsklasse und der Thread-Priorität abgeleitete Basispriorität für das Scheduling relevant ist. Das reine Prioritätsscheduling von Windows XP konnte ebenfalls durch die Simulation verschiedener Laststufen mit variierenden relativen Basisprioritäten bewiesen werden. Ein Thread mit niedriger Basispriorität bekommt nur dann Rechenzeit, wenn in einer höheren Basispriorität keine rechenbereiten Threads mehr vorhanden sind.

Im Kapitel 6 werden die Auswirkungen der Definition von Affinitätsmasken analysiert. Die Ergebnisse zeigen klar, dass durch eine Affinitätsmaske verhindert werden kann, dass ein Prozess oder dessen Threads bestimmte CPUs verwenden. Dies vermindert aber in den allermeisten Fällen nur die maximale Skalierung auf der Hardware. Das setzen eines „Ideal-Prozessor“ über die Windows API konnten wir auf Java-Ebene nicht testen. Dies wäre aber möglicherweise die bessere Methode als harte Affinitäten da im Bedarfsfall trotzdem auf „nicht-ideale“ Prozessoren ausgewichen werden kann was bei einer gesetzten Affinitätsmaske nicht mehr möglich ist.

Die restlichen Testcases befassen sich mit der Thread-Skalierung (siehe Kapitel 7). Wir konnten belegen, dass Java-Threads unter Windows XP 1:1 auf Kernel-Threads abgebildet werden und somit eine parallele Abarbeitung auf mehreren Recheneinheiten durch den Betriebssystem-Scheduler möglich wird. Anschliessend wurde die Skalierung der Thread-Anzahl durch Messungen mit 1, 2, 8, 32, 128 und 512 Threads jeweils mit einem oder zwei Prozessoren durchgeführt. Die Messungen haben gezeigt, dass die Anwendung annähernd linear mit der Anzahl der Threads skaliert (ohne Synchronisierung). Mit zugeschalteter Synchronisation konnte stieg offenbar der Verwaltungsaufwand was auf einem Single-CPU System zu einem Performance-Einbruch bei einer hohen Thread-Anzahl führte. Das Dual-CPU System zeigte sich davon deutlich weniger beeindruckt. Auf beiden Systemen liess sich durch Optimierung der Synchronisierung (lock partitioning / lock striping / CAS) die Skalierung wieder bis zur annähernden Linearität verbessern.

Die Ergebnisse sprechen eine deutliche Sprache. Um parallele Hardware optimal nutzen zu können ist eine Verteilung von aufwändigen Berechnungen auf mehrere Threads unverzichtbar. Bei entsprechender Programmierung kann die Anzahl der Threads dabei auch ohne Performance-Einbrüche massiv höher liegen als die Anzahl verfügbarer CPUs. Hier ist allerdings eine durchdachte Architektur unabdingbar. Bei unglücklicher Verwendung von Locks kann schnell ein Teil der gewonnenen Performance durch Blockierungen zunichte gemacht werden. Allgemein gilt aber, dass durch die Verwendung von Threads auf Multi-CPU Systemen eine Vervielfachung der Rechenkapazität erreicht werden kann und auf Single-CPU Systemen kaum zu Nachteilen führt (bei geschickter Synchronisierung). Optimal ist hierbei natürlich eine Architektur, welche eine variable Anzahl Threads erlaubt um die Anwendung optimal auf die Hardware anzupassen.

2. Inhaltsverzeichnis

1. Management Summary ... 3
2. Inhaltsverzeichnis ... 4
3. Dokumentinformationen ... 5
3.1. Referenzierte Dokumente ... 5
3.2. Definitionen und Abkürzungen ... 5
3.3. Links ... 5
4. Einleitung ... 6
5. Testcases Prioritäten ... 7
5.1. Interpretation Testcase 3 ... 7
5.2. Interpretation Testcase 4 ... 7
5.3. Interpretation Testcase 5 ... 8
5.4. Interpretation Testcase 6 ... 8
5.5. Empfehlungen ... 9
6. Testcases Affinität ... 10
6.1. Interpretation Testcase 7 ... 10
6.2. Emfehlungen ... 11
7. Testcases Skalierung ... 12
7.1. Interpretation Testcase 2 ... 12
7.2. Interpretation Testcase 1 ... 12
7.3. Interpretation Testcase 8 ... 13
7.4. Interpretation Testcase 9 ... 14
7.4.1. Methodensynchronisation ... 14
7.4.2. Lock paritioning ... 16
7.4.3. CAS ... 16
7.5. Empfehlungen ... 18
7.6. Interpretation Testcase 10 ... 19
8. Einsatzgebiete ... 20
9. Allgemeine Bemerkungen ... 21
10. Abschliessende Bemerkungen ... 22
11. Glossar ... 23
12. Verzeichnisse ... 25
12.1. Tabellenverzeichnis ... 25
12.2. Abbildungsverzeichnis ... 25
12.3. Code Listings ... 25
12.4. Index ... 25

3. Dokumentinformationen

3.1. Referenzierte Dokumente

Tabelle 1 Referenzierte Dokumente

Referenz	Beschreibung
[1]	Basisanalyse
[2]	Software Test Document (STD)
[3]	Software Design Document (SDD)
[4]	Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1

3.2. Definitionen und Abkürzungen

Tabelle 2 Abkürzungen

Abkürzung	Beschreibung
AMD	Advanced Micro Devices
API	Application Programming Interface
CAS	Compare And Swap / Compare And Set
CPU	Central Prozessing Unit (Hauptprozessor)
HDTV	Hight Definition Television
JOMP	Java OpenMP
TBB	Thread Building Blocks

3.3. Links

Tabelle 3 Links

Referenz	Beschreibung
[CODEANALYST]	AMD CodeAnalyst: http://developer.amd.com/cawin.jsp
[PARALLELSKAL]	Microsoft, Verwenden der Parallelität für Skalierbarkeit: http://www.microsoft.com/germany/msdn/library/net/ ↵ VerwendenDerParallelitaetFuerSkalierbarkeit.mspx

4. Einleitung

Dieses Dokument bietet Platz für Schlussfolgerungen, Interpretationen, weiterführende Erklärungen und in gewissen Fällen eventuell Spekulationen. Hierbei geht es im Wesentlichen um eine erweiterte Betrachtung der festgehaltenen Testresultate in [2].

Das Dokument ist nach den Haupteinflussbereichen der Priorität, Affinität und Skalierung gegliedert. Ziel ist es für jeden Bereich Aussagen über markante Messwerte und deren Einfluss auf die Praxis der parallelen Programmierung herauszuheben. Dabei wird versucht die Hintergründe der Messergebnisse zu beleuchten um die Zusammenhänge besser verstehen zu können.

Wo dies möglich und sinnvoll ist werden entsprechende Empfehlungen zur Implementierung gegeben sowie Hinweise auf mögliche Probleme festgehalten.

5. Testcases Prioritäten

Die Testcases 3, 4, 5 und 6 geben einen Einblick in die Prioritätsverwaltung von Java Threads unter Windows XP. Die durchgeführten Tests legen die Wirkung der Java Thread-Prioritäten auf allen Ebenen offen. Dies schliesst die Offenlegung der Abbildung von Java-Prioritäten auf Kernel-Prioritäten ebenso ein wie die Analyse des Programmverhaltens unter verschiedenen Prioritäts- und Laststufen.

5.1. Interpretation Testcase 3

Hier zeigt sich ein sehr interessantes Bild bei der Abbildung der Java Thread Prioritäten auf Win32 Prioritäten. Wie in [1] (Kapitel 6) erwähnt veraltet Windows XP sieben Thread Prioritäten (`IDLE`, `LOWEST`, `BELOW_NORMAL`, `NORMAL`, `ABOVE_NORMAL`, `HIGHEST`, `TIME_CRITICAL`). Java verwendet ein Modell mit 10 Prioritätsstufen (1 bis 10). Diese werden aber nicht mathematisch auf die 7 Win32 Thread Prioritäten abgebildet sondern nur auf deren 5. Dies führt dazu, dass mehrere Java-Prioritäten auf die gleiche Win32 Priorität abgebildet werden. So bekommen Threads mit der Java-Priorität 5 und 6 dieselbe Win32 Basispriorität von 8.

Des Weiteren heisst dies für Java Threads, dass die Prioritäten `IDLE` und `TIME_CRITICAL` nicht ausgewählt werden können. Der Bereich der damit erreichbaren Win32 Basisprioritäten erstreckt sich somit von 6 bis 10. Um höhere bzw. tiefere Werte erreichen zu können muss die Prioritätsklasse des Prozesses verändert werden. Mehr dazu unter Interpretation Testcase 4.

5.2. Interpretation Testcase 4

Tabelle 4 Bereich der Basisprioritäten in Abhängigkeit der Prozessprioritätsklasse

Process priority Class	Tiefste Basispriorität	Höchste Basispriorität
`IDLE_PRIORITY_CLASS`	2	6
`BELOW_NORMAL_PRIORITY_CLASS`	4	8
`NORMAL_PRIORITY_CLASS`	6	10
`ABOVE_NORMAL_PRIORITY_CLASS`	8	12
`HIGH_PRIORITY_CLASS`	11	15
`REALTIME_PRIORITY_CLASS`	22	26

Der Bereich der Prozessprioritäten `HIGH` und `REALTIME` liegt überproportional über dem Bereich der anderen Prioritäten. Es ist keine Überlappung bei der Prozesspriorität `REALTIME` möglich. Somit kann kein Thread im Prozesskontext mit einer tieferen Priorität eine höhere Einstufung erreichen als ein Thread im Prozesskontext eines `REALTIME` Prozesses.

Über die Java-API ist es nicht möglich die Prozessprioritäten zu beeinflussen. Diese wird also ohne Änderung von aussen auf `NORMAL` festgesetzt. Allerdings kann mittels der Thread-Priorität eine Basis-Priorität erreicht werden, die der Standardpriorität von `ABOVE_NORMAL` bzw. `BELOW_NORMAL` entspricht. Die Basispriorität kann im Bereich von 6 bis 10 angepasst werden. Die Basispriorität 6 entspricht der Standardpriorität für Threads in einem Prozesskontext mit der Priorität `BELOW_NORMAL`. Die Basispriorität 10 entspricht der Standardpriorität für Threads in einem Prozesskontext mit der Priorität `ABOVE_NORMAL`.

5.3. Interpretation Testcase 5

Wie unter Interpretation Testcase 4 beschrieben gibt es aufgrund der Überschneidungen bei den Prioritäten mehrere Möglichkeiten auf dieselbe Basispriorität zu kommen. Im Test wird das Verhalten eines Threads mit der Java-Priorität 5 im Kontext eines Prozesses mit der Prioritätsklasse `NORMAL` dem Verhalten eines Threads mit der Java Priorität 2 im Kontext eines Prozesses mit der Prioritätsklasse `ABOVE_NORMAL` gegenübergestellt. Beide Male resultiert eine effektive Basispriorität von 8. Siehe dazu auch die in Testcase 4 erstellte Prioritätstabelle.

Der Versuch hat gezeigt, dass im Endeffekt nur die Basispriorität das Scheduling Verhalten beeinflusst. Threads eines Prozesses mit höherer Priority Class werden bei gleicher Basispriorität nicht bevorzugt behandelt.

Für den Programmierer heisst dies, dass die Anhebung der Prozesspriorität nicht unbedingt in einer höheren Prioritätsstufe (Basispriorität) als die Threads eines Prozesses mit tieferer Prozesspriorität resultieren muss. Falls die Threads des Prozesses mit tieferer Prioritätsstufe die höchstmögliche Basispriorität besitzen kann diese durchaus höher sein als die Standardpriorität der Threads im Prozesskontext des höher priorisierten Prozesses.

Schön zu sehen ist auch, dass die konsumierte CPU-Zeit bei allen Messungen konstant bleibt (unabhängig von der Laststufe). Dies erscheint auch logisch, da die CPU-Zeit nur die Anzahl „verbrauchter" CPU-Zyklen widerspiegelt. Die Berechnung des Bildes ist immer gleich aufwändig und benötigt eine konstante Anzahl Prozessor-Zyklen. Abhängig von der Anzahl Kontextwechsel oder anfallendem Locking bzw. I/O Wait kann die CPU-Zeit leicht schwanken, sollte sich aber nicht im Bereich von Faktoren ändern.

5.4. Interpretation Testcase 6

Hier wurden Vergleiche mit unterschiedlicher (resultierender) Basispriorität angestellt (bei gleich bleibender Prozess-Priorität). Die Versuche wurden jeweils mit der Basispriorität 6 bzw. 8 unter verschiedenen Laststufen (keine Last, 1 Thread mit Basispriorität 8 sowie 2 Threads mit Basispriorität 8) durchgeführt.

Windows arbeitet mit einem reinen Priority-Scheduling. Dies bedeutet, dass Threads einer tieferen Prioritätsstufe nur dann Rechenzeit bekommen, wenn kein Thread einer höheren Prioritätsstufe rechenbereit ist.

Daraus resultiert, dass ohne Last in höheren Prioritätsstufen die gesamte Rechenzeit auch an tiefer priorisierte Threads verteilt wird.

Die im Test verwendete Last besteht aus einem einzigen Calculator-Thread. Bei tiefer Last wird ein einziger Calculator Thread gestartet der sich auf einer CPU konzentriert. Somit bleibt die zweite CPU für die Java-Anwendung frei. Aus diesem Grund entspricht die Laufzeit einem Single-CPU Testlauf.

Im Falle einer grossen Last (2 Calculator, 2 Threads) werden beide Threads auf die verfügbaren CPUs verteilt wobei keine Rechenzeit für die Berechnung mit tieferer Basispriorität zur Verfügung gestellt wird. Im Test wird der Java-Berechnung nur CPU Zeit zugewiesen wenn der Calculator-Thread nicht rechenbereit ist (z.B. blockiert bei I/O Zugriffen).

Bei höherer, relativer Priorität der Java-Berechnung bezogen auf die Last tritt genau der gegenteilige Fall ein. Die Calculator-Threads bekommen nur dann Rechenzeit, wenn die Threads der Java-Berechnung nicht rechenbereit sind (z.B. blockiert durch I/O Operationen).

5.5. Empfehlungen

Auf Ebene Java kann der Programmierer nur innerhalb der 10 Java Prioritäten Einfluss auf die Basispriorität nehmen. Generell eignen sich diese Prioritäten gut um innerhalb der eigenen Anwendung Priorisierungen vorzunehmen. Global gesehen ist der Einsatz aber schon deutlich eingeschränkt da die Einsatzumgebung auf einem Desktop-Rechner unbekannt ist. Das heisst, dass häufig weder die Anzahl weiterer Prozesse und Threads sowie deren Rechenzeit-Bedarf oder gar deren Prioritäten bekannt sind.

Java arbeitet beispielsweise bereits automatisch mit verschiedenen Prioritäten. So haben Daemon Threads für interaktive Elemente wie die Ereignisverwaltung für GUIs bereits eine Java-Priorität von 6 um eine schnelle Reaktion auf Benutzeraktionen zu erlauben. Leider belegt unser Prioritäts-Mapping aus Testcase 4, dass sowohl die Java Priorität 5 und 6 auf die selbe Basispriorität abgebildet werden. Deshalb scheint diese Konfiguration unter Windows keinen Einfluss zu zeigen. Unter Windows ist es also empfehlenswert aufwändige Berechnungen, welche die Reaktionsgeschwindigkeit des GUIs nicht beeinflussen sollen, in Threads mit der Java-Priorität von 4 oder geringer zu verlagern. Der Nachteil dieser Methode liegt dann natürlich darin, dass diese Threads die Basispriorität von 7 oder geringer bekommen und somit auch im Konkurrenzkampf mit anderen Anwendungen (Standard-Basispriorität: 8) das Nachsehen haben.

6. Testcases Affinität

Testcase 7 befasst sich mit der manuellen Konfiguration einer Affinitätsmaske über ein externes System-Tool und deren Auswirkungen auf die Skalierung. Dabei werden die Auswirkungen sowohl mit als auch ohne simulierte externe Last analysiert.

6.1. Interpretation Testcase 7

Ohne Affinität können alle Threads auf alle verfügbaren CPUs verteilt werden. Das System (Windows XP) strebt eine gleichmässige Auslastung aller Rechnwerke an. Bei zwei CPUs und einem einzigen Thread bedeutet dies, dass Windows versucht auf beiden CPUs 50% der Berechnungen durchzuführen. Dabei wird der Threads von CPU zu CPU verschoben (Time-Slicing):

Abbildung 1 Verteilung einer Single-Thread Anwendung auf zwei CPUs

Die Abbildung wurde vom AMD CodeAnalyst [CODEANALYST] erstellt und zeigt die Verteilung einer Anwendung mit einem einzigen Thread auf zwei CPUs. Die daraus resultierende CPU-Gesamtlast beträgt 50%. Die Anwendung läuft dadurch genau so schnell wie wenn nur ein einziger CPU zur Verfügung stehen würde.

Bei gleich vielen Threads wie CPUs können alle CPUs ausgelastet werden:

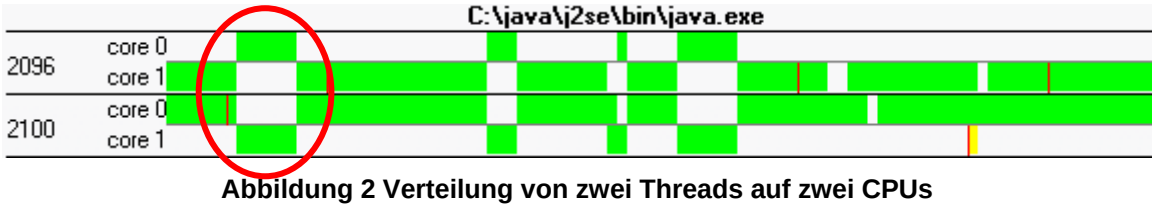

Abbildung 2 Verteilung von zwei Threads auf zwei CPUs

Auch diese Abbildung wurde mit dem AMD CodeAnalyst erstellt und zeigt unsere Mandelbrot-Applikation bei der Arbeit. Interessant ist hierbei insbesondere, dass der Windows Scheduler hier offenbar trotzdem versucht jeden Thread auf jeder CPU auszuführen. Dies zeigt sich daran, dass die Threads offenbar zeitweilig ihre „Plätze“ tauschen und auf dem anderen CPU weiterarbeiten. Dies erklärt auch die massiv höhere Anzahl Kontextwechsel auf Multi-CPU/Multi-Core Systemen, die wir in Testcase 1 (siehe Kapitel 7.2) ermittelt haben. Trotzdem beträgt die CPU Gesamtlast hier zu jedem Zeitpunkt 100% da beide Prozessoren zu jedem Zeitpunkt mit Berechnungen beschäftigt sind.

Beim setzen einer Prozessaffinität wird diese auch an die Threads weitervererbt. Durch die Affinität werden die Threads dieses Prozesses auf die ausgewählten CPUs konzentriert. Diese können nun nicht mehr auf einen nicht explizit zugewiesenen Prozessor wechseln. Dies bedeutet auch, dass ein Thread auf einer bereits ausgelasteten CPU nicht auf einen alternativen Prozessor ausweichen kann. Nur Threads ohne Affinität können durch den Scheduler beliebig auf andere CPUs verteilt werden. Dies belegt die nächste Grafik:

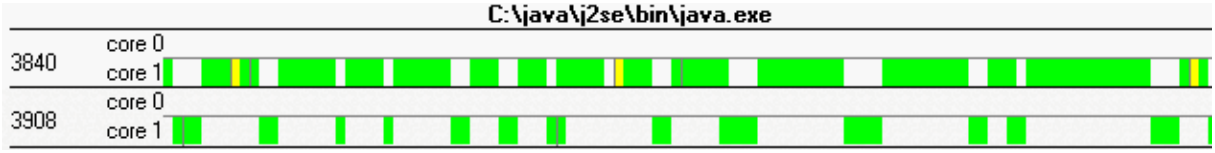

Abbildung 3 2 Threads mit gesetzter CPU Affinität auf CPU0, ohne Last

Die zwei rechnenden Threads der Mandelbrot Anwendung konkurrieren nun um CPU1, da durch die Affinitätsmaske der Wechsel auf CPU0 unterbunden wurde. Derselbe Effekt tritt ein, wenn die Anwendung nicht alleine um CPU1 konkurriert. In Testcase 7 haben wir zusätzlich noch einen Calculator ohne Affinität laufen lassen. Dieser hat nun unserer Mandelbrot Applikation einzelne Zeitscheiben auf CPU1 weggenommen (obwohl CPU0 frei gewesen wäre). Erst die manuelle Festlegung der Affinität

des Calculators auf die freie CPU sorgte dafür, dass diese beide voll ausgelastet wurden und unsere Anwendung eine für sich alleine hatte.

Durch die Definition von komplementären Affinitätsmasken lassen sich mehrere Prozesse auf unterschiedliche CPUs verteilen. Die Last kann so gezielt konzentriert werden. Dabei muss aber beachtet werden, dass weitere Prozesse (ohne bzw. mit der gleichen Affinitätsmaske) auch um die verfügbaren Prozessoren konkurrieren können. Ausserdem ist eine übliche Umgebung nicht dermassen überschaubar wie unsere Testumgebung. Die manuelle Festlegung der Affinitäten funktioniert nur dann so gut, wenn die laufenden Anwendungen und Threads bekannt sind. Dynamisch gestartete Dienste oder Anwendungen müssten dabei ihre Affinitätsmaske selber setzen oder sie müsste bei jedem start manuell neu gesetzt werden. Ausserdem ist so nur eine 100% Auslastung zu erreichen wenn beide Anwendungen jeweils eine ganze CPU auslasten können. Würde der Calculator nur 20% der CPU-Zeit einer CPU benötigen, dann könnten auch die restlichen 80% nicht von der Mandelbrot Anwendung genutzt werden, da dieser der Wechsel aufgrund der Affinitätsmaske nicht mehr erlaubt ist.

6.2. Emfehlungen

Affinitätsmasken machen unserer Meinung nach nur ganz selten Sinn. Normalerweise nimmt man dem System dadurch lediglich die Möglichkeit das volle Potential der Hardware zu nutzen. In unserem Fall wurde die maximal verfügbare CPU-Leistung für unsere Mandelbrot-Applikation auf 50% eingeschränkt. Dabei müssen diese 50% zusätzlich noch mit anderen Prozessen/Threads von anderen Applikationen geteilt werden. Im schlimmsten Fall kann es passieren, dass der zugewiesene Prozessor ausgelastet ist und eine freie CPU aufgrund der Affinitätsmaske nicht genutzt werden kann.

In diesem Rahmen möchten wir nochmals auf die in der Win32 API vorhandene Möglichkeit einen „Ideal-Processor“ zu definieren verweisen. Diese API ermöglich es dem Betriebssystem mitzuteilen auf welchem Prozessor die Applikation Idealerweise laufen soll. Ist die ideale CPU dabei gerade nicht verfügbar so wird die Anwendung aber auch auf einen „nicht-idealen“ (alternativen) Prozessor verlagert. Dies ist immer noch die bessere Möglichkeit als die Anwendung warten zu lassen. Frei nach dem Motto „eine langsame Ausführung ist besser als gar keine“. Die Möglichkeit einen idealen Prozessor zu definieren kann insbesondere bei NUMA (siehe Erklärung in [1]) Systemen einen Performance-Zuwachs ergeben, da der ideale Prozessor auf solchen Systemen üblicherweise derjenige ist, auf dem die verwendeten Daten lokal verfügbar sind. Mangels der Unterstützung dieser API in Java konnten wir dies nicht näher testen. Sinnvoll wäre hier ein Vergleich der Remote-Memory Zugriffe mit bzw. ohne gesetzte „Ideal-Processor“ Maske bei einer speicherintensiven Anwendung. Der AMD CodeAnalyst (siehe [CODEANALYST]) kann hier bei der Analyse helfen. In Abbildung 1 sind deutlich rote Markierungen zu erkennen. Diese markieren laut CodaAnalyst „Non-Local Memory Access“ und stellen somit ein Optimierungspotential dar.

7. Testcases Skalierung

In diesem Kapitel werden die Testcases diskutiert, welche die direkte Analyse der Skalierung zum Ziel haben. In Testcase 2 werden die zur Thread-Skalierung notwendigen Analysen zur Abbildung von Java-Threads auf Kernel-Threads gemacht. Testcase 1 analysiert dann ob die Threads auch wirklich auf mehrere Prozessoren verteilt werden können. In Testcase 8 und 9 der Einfluss einer variierenden Anzahl Threads auf die Skalierung jeweils ohne Locking bzw. mit verschiedenen Locking-Techniken untersucht.

7.1. Interpretation Testcase 2

Die Ergebnisse dieser Testreihe sind eindeutig und belegen eine 1:1 Abbildung von Java-Threads auf Win32 Threads. Das heisst, dass für jeden Java-Thread ein entsprechendes äquivalent auf im Betriebssystem-Kernel existiert. Dass selbst bei null (0) selbst erzeugen Worker-Threads eine gewisse Anzahl Threads existieren (in unseren Beispiel 13) liegt an der internen Architektur. Auch das Hauptprogramm (`main()`) läuft innerhalb eines Threads ab. Die anderen Threads sind im Hintergrund laufende Daemon-Threads wie beispielsweise die Garbage-Collection oder Threads zur GUI Ereignisbehandlung. Die Anzahl der Daemon-Threads kann von Anwendung zu Anwendung variieren. So kann die Verwendung von JVM Klassen zur Erzeugung weitere Daemon-Threads führen. Beispielsweise führt erst die Instanzierung von GUI-Klassen zur Erzeugung von Threads zur Ereignisbehandlung.

Es ist aber ganz klar zu sehen, dass jeder erzeugte Java-Thread auch im Kernel abgebildet wird. Die Sun Java HotSpot VM 1.5 arbeitet hier also mit einer 1:1 Abbildung. Anhand der gleich bleibenden Prozess-ID (PID) kann auch erkannt werden, dass die Anwendung während des Tests nicht neu gestartet wurde. Die Threads wurden also zur Laufzeit erstellt.

Offensichtlich scheint auch ein Grenzwert zu existieren. Auf unserem System (Windows XP Pro 32 Bit, Sun HotSpot 1.5 VM, 4GB RAM) lag diese Grenze bei 7146 Threads. Jeder weitere Thread führte zu einer „Out of Memory Execption“ und zwar unabhängig von einem eventuell vorhandenen `-Xmx` Parameter um mehr maximalen Speicher für den Heap zu erlauben.

7.2. Interpretation Testcase 1

Die Berechnungsdauer liess sich auf unserem Multi-Prozessor System durch den Einsatz von zwei Threads auf die hälfte reduzieren. Eine Single-Threaded Anwendung hätte hier also nur maximal 50% der vorhandenen Rechenleistung genutzt.

Auf den ersten Blick mag es verwirrend sein, dass die CPU-Zeit weder von der Anzahl Threads noch von der Anzahl CPUs abhängig ist und konstant bleibt. Dies ist aber nur logisch, da der Berechnungsaufwand für das gesamte Bild bei jedem Testdurchlauf identisch ist. Der Aufwand wird bei 2 CPUs und 2 oder mehr Threads lediglich auf zwei Rechenwerke verteilt. Die Zusammenhänge werden etwas klarer, wenn man die aktuell verbrauchte CPU-Zeit in einem Tool wie dem Task Manager oder dem Process Explorer anzeigt. Dort zählt die Spalte CPU-Zeit nämlich doppelt so schnell hoch bei zwei CPUs und zwei Threads. Pro Sekunde werden also 2 Sekunden CPU-Zeit „verbraucht“.

Ein interessanter Aspekt ist auch die Anzahl der Kontextwechsel. Diese steigen nämlich auf unsererer 2 CPU Maschine sprunghaft an und dies unabhängig davon ob 1 oder 2 Threads verwendet werden. Wie im Kapitel 6 erwähnt versucht Windows durch ständiges weiterreichen der Threads auf alle verfügbaren CPUs eine gleichmässige Auslastung zu erreichen. Auf unserem System führt dies dazu, dass bei der Verwendung von nur einem einzigen Thread beide CPUs nahezu exakt 50% ausgelastet werden (resultierende Gesamtlast: 50%). Die andauernden CPU Sprünge führen zu der erhöhten Anzahl Kontextwechsel.

Auch wir dachten zuerst, dass bei der Verwendung von 2 Threads die Kontextwechsel zurückgehen würden, da es kaum Sinn macht zwei rechenbereite Threads untereinander abzutauschen. Wie in Kapitel 6 aber ebenso zu sehen ist werden zwei Thread teilweise trotzdem untereinander ausgetauscht und laufen auf dem jeweils anderen CPU weiter.

Selbst nach längeren Online-Recherchen konnten wir noch keinen Kernel-Parameter finden um dieses Verhalten zu beeinflussen. Grundsätzlich scheint eine erhöhte Anzahl Kontextwechsel aber auch keinen merklichen Einfluss auf die Performance zu haben. Weder die Berechnungsdauer noch die benötigte CPU-Zeit steigen dadurch merklich an. Lediglich NUMA (siehe Erklärung in [1]) Systeme könnten durch dieses Verhalten negativ beeinflusst werden da so im Schnitt 50% der Speicherzugriffe auf Remote-Speicher durchgeführt werden müssen was zu einer erhöhten Buslast führen kann.

7.3. Interpretation Testcase 8

Testcase 8 hat gezeigt, dass bei Einzelprozessorsystemen die Anzahl Threads keine nennenswerte Rolle spielt sofern keine Synchronisation stattfinden muss. Die Berechnungszeit verändert sich zwischen einem und 512 Threads kaum. Die Schwankungen gehen in der Messungenauigkeit unter.

Bei zwei CPUs sieht das Resultat nicht viel anders aus. Weder die CPU Zeiten noch die Berechnungszeit verändern sich messbar. Lediglich zwischen der Konfiguration mit einem bzw. zwei Threads ist eine Halbierung der Berechnungszeit feststellbar. Dies liegt darin begründet, dass die Aufgabe nun auf beide Prozessoren verteilt wird. Die Gesamtauslastung steigt dabei von 50% (1 Thread, 1 CPU ausgelastet) auf 100% (2 Threads, 2 CPUs ausgelastet). Auch hier scheint der zusätzliche Verwaltungsaufwand von bis zu 512 Threads keinen nennenswerten Einfluss auf die Berechnungszeit zu haben. Zu beachten ist hierbei aber auch, dass die Messungen ohne Synchronisierungsaufwand gemacht wurden. Tests mit Synchronisierung folgen in Testcase 9 (siehe auch Kapitel 7.4).

Die Frage wo hier die optimale Anzahl Threads liegt ist nicht einfach zu beantworten. In unserem Beispiel können bereits zwei Threads beide CPUs voll auslasten. Sind diese aber beispielsweise teilweise durch I/O Operationen blockiert kann es sinnvoll sein, dass die Anzahl Threads höher gewählt wird als die Anzahl CPUs. Dazu kann die folgende Formel aus [PARALLELSKAL] helfen:

```
NumThreads = NumCPUs / (1 – BP)
```

Wobei BP den Prozentsatz der Zeit darstellt in dem die Threads blockiert sind (Blocked-Percentage). Verbringen die Threads also 25% im blockierten Zustand so kann es bei einem 2 CPU System sinnvoll sein 3 Threads (2/(1-0.25) = 2.66) einzusetzen.

Leider ist diese Formel nur bedingt praxistauglich da einerseits der Prozentsatz der Blockierten Zeit häufig nicht so einfach vorhersehbar ist und andererseits der Grad der tatsächlich verfügbaren parallelen Einheiten nicht einbezogen wird. Bei HyperThreading CPUs stehen nicht alle Prozessorteile mehrfach zur Verfügung. Dies führt unter Umständen zu weiteren Blockierungen. Auch kann der Einsatz weiterer Threads die Speicherbandbreite weiter belasten und zu längerer Blockierung der Threads führen.

Klar ist aber, dass durch fehlende Parallele Verarbeitung schlicht nur `1/AnzahlCPUs` Prozent der verfügbaren Kapazität genutzt werden kann. Bei unserem 2 CPU System kann ohne Parallelisierung somit nur 50% der Gesamtleistung genutzt werden.

Die oben stehende Formel bestätigt im umgekehrten Sinne auch, dass nie weniger Threads als CPUs verwendet werden sollten um eine optimale Skalierung zu erreichen (Angenommene Blockierung: 0%). Dieser Fall entspricht auch ziemlich genau unserer Testreihe ohne Synchronisierung.

7.4. Interpretation Testcase 9

7.4.1. Methodensynchronisation

Hier haben wir wie in [3] beschrieben eine Methodensynchronisation verwendet um die Auswirkungen von langen Synchronisierungsbereichen zu zeigen. Der Nachteil liegt hier darin, dass nur ein Thread einen Pixel schreiben kann und alle anderen warten müssen bis der schreibende damit fertig ist. Dadurch wird der Bereich der Synchronisation relativ lang und die Threads treten häufig in die Region ein und verlassen diese wieder (bei jedem Pixel einmal). Bei den über 700'000 Schreibzugriffen für ein gesamtes Bild (1016x718 Pixel) dürfte der Aufwand für das Locking spürbar werden. Zur Erinnerung hier nochmals die synchronisierte Methode (aus [3]):

```
public synchronized void setRGB(int startX, int startY, int w,
                int h, int[] rgbArray, int offset, int scansize) {
    count++;
    super.setRGB(startX, startY, w, h, rgbArray, offset, scansize);
}
```

Listing 1 Methodensynchronisation

Nachfolgend die Ergebnisse:

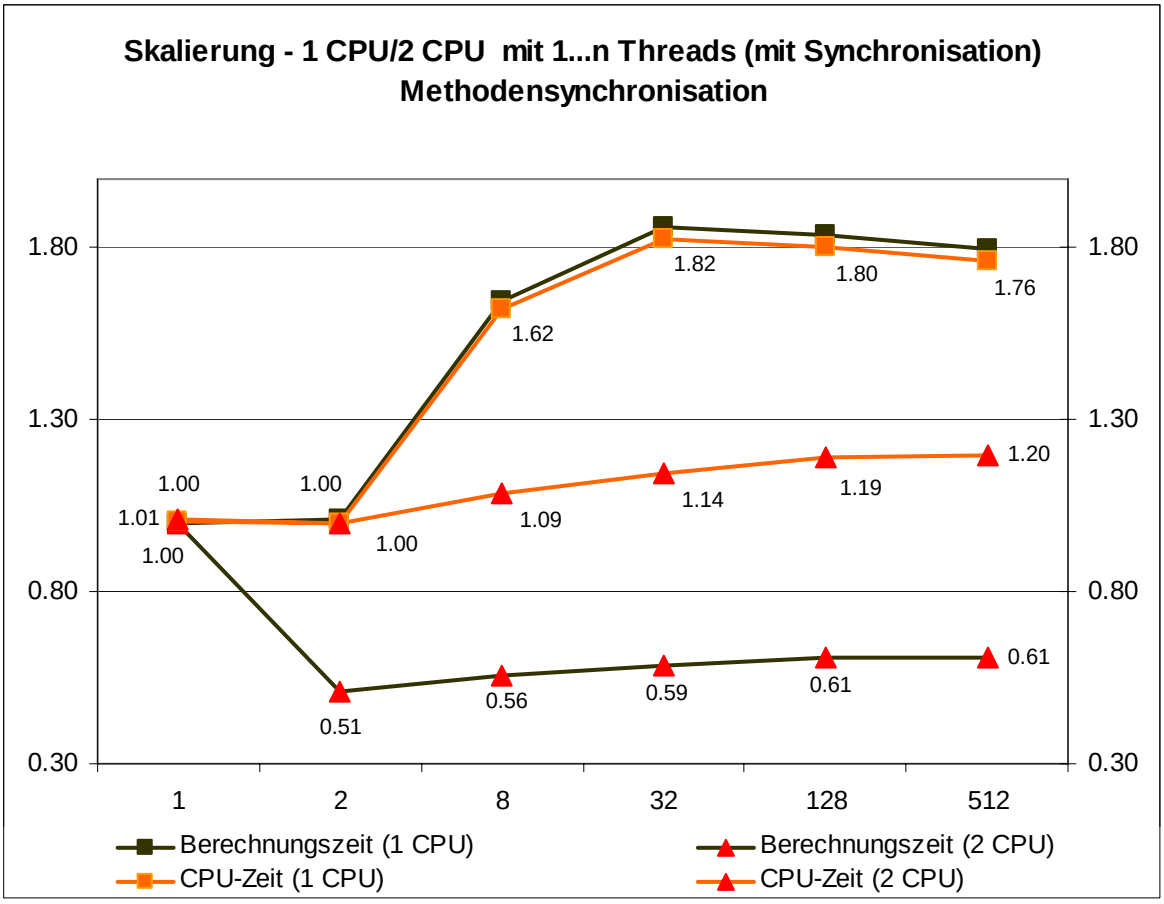

Abbildung 4 Skalierung 1 CPU/2 CPU mit Methodensynchronisation

Erstaunlich ist der unterschiedliche Verlauf zwischen 1 und 2 CPU Konfiguration. Auf dem Einzelprozessorsystem zeigt sich das erwartete Verhalten. Je mehr Threads um den Lock konkurrieren (lock contention) umso mehr Verwaltungsaufwand fällt an. Dieser belastet die CPU zusätzlich und lässt sowohl die CPU-Zeit als auch die Berechnungszeit sprunghaft ansteigen. Auf der Dual-CPU Konfiguration zeigt sich der Verlauf aber weitgehend unbeeindruckt vom zusätzlichen Aufwand und das ob-

wohl hier eigentlich aufgrund der tatsächlich parallelen Verarbeitung häufiger konkurrierende Lock-Anfragen zu erwarten sind.

Eine eindeutige Erklärung dafür konnten wir leider noch nicht finden. Sicher ist aber, dass bei zwei Prozessoren die Wahrscheinlichkeit, dass ein Thread der den Lock gerade hält und unterbrochen wurde wieder Rechenzeit erhält, doppelt so hoch ist. Die Folgende Grafik zeigt den logischen Arbeitsablauf der Threads und das Scheduling:

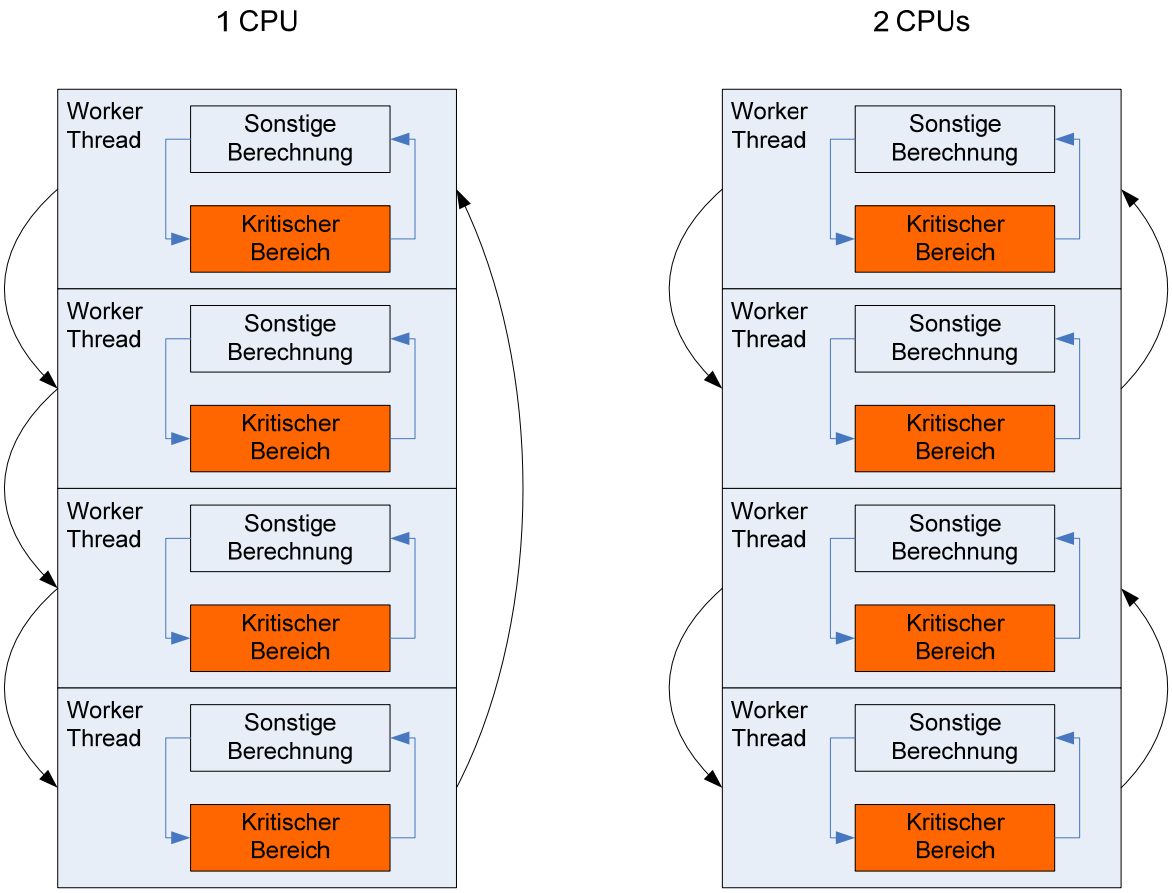

Abbildung 5 Thread-Abarbeitung

Die Grafik zeigt ein simples Round-Robin Scheduling. Ist nur ein CPU verfügbar wird dieser jedem einzelnen Thread in Reihenfolge zugewiesen. Im ungünstigsten Fall wird der erste Thread im kritischen Bereich unterbrochen und behält somit den Lock. Die folgenden Threads erhalten dann nacheinander die Rechenzeit, werden aber spätestens beim Eintritt in den kritischen Bereich blockiert. Sind alle Threads durchlaufen bekommt der erste wieder einen Zeitschlitz. Verlässt dieser dann den kritischen Bereich bis zum Ende des Zeitschlitzes nicht, so bekommen wieder alle Threads einen Zeitschlitz und dies nur um zu prüfen ob der Lock frei ist um dann gleich wieder zu blockieren. Angenommen wir arbeiten mit 512 Threads so wird in diesem Fall 511 Mal geprüft ohne, dass ein Thread wirklich weiterkommen würde.

Im Fall von 2 CPUs kann logisch von einer Zweiteilung der Threads ausgegangen werden. Ein Thread bekommt also durchschnittlich doppelt so häufig einen Zeitschlitz und verlässt den kritischen Bereich deshalb potentiell schneller. Dabei ist zu beachten, dass bei der vorliegenden Methodensynchronisation der kritische Bereich nicht wie die Threads in zwei logischen Gruppen gesehen werden können. Befindet sich einer der vier abgebildeten Threads im kritischen Bereich so ist dieser für die verbleibenden drei Threads gesperrt. Das (ineffiziente) durchprobieren aller Threads ob sie den Lock bekommen können kann hier durch die Aufteilung auf zwei CPUs ebenfalls doppelt so schnell geschehen was dazu führt, dass derjenige Thread, der den Lock hält, schneller weitermachen kann um den kritischen Bereich zu verlassen.

Die Dauer für die ein Lock gehalten wird ist eine sehr kritische Grösse weil es die Wahrscheinlichkeit erhöht, dass dadurch andere Threads blockiert werden.

7.4.2. Lock paritioning

Durch „lock partitioning“ und „lock striping“ (siehe [1]) wird versucht die Locks so aufzuteilen, dass verschiedene kritische Bereiche entstehen. In Abbildung 1 existiert nur ein Lock für alle kritische Bereiche was einem wechselseitigen Ausschluss zwischen allen Threads entspricht. In unserer Anwendung wäre die Verwendung von zwei Locks für die obere und untere Bildhälfte denkbar. Somit wären durchschnittlich nur die hälfte der Threads von einem gesperrten Lock betroffen was einer Halbierung der Blockierungswahrscheinlichkeit entsprechen würde. Im Beispiel für die Objektsynchronisation sind wir noch einen Schritt weiter gegangen und haben einen separaten Lock für jeden einzelnen Pixel erstellt. In diesem Fall schützt jeder Lock nur einen einzelnen Pixel vor konkurrierendem Zugriff. Da nie zwei Threads gleichzeitig auf denselben Pixel schreiben schlägt eine Lock-Anforderung nie fehl. Anhand der Ergebnisse vermuten wir sogar, dass die Sun HotSpot VM 1.5 diesen Lock bei der Abarbeitung sogar entfernt da er gar nicht benötigt wird.

Allerdings haben wir sogar bei diesem „feinen“ Objektlocking einen globalen Lock eingebaut den die JVM nicht entfernen kann: Den gemeinsamen Zugriff auf eine Counter-Variable. Hier zur Erinnerung der verwendete Code:

```
    public void setRGB(int startX, int startY, int w, int h,
                       int[] rgbArray, int offset, int scansize) {
        synchronized (this) {
            count++;
        }
        synchronized (locks[startX][startY]) {
            super.setRGB(startX, startY, w, h, rgbArray,
                         offset, scansize);
        }
    }
```

Listing 2 Code für feines Locking (Objektsynchronisation)

Die Synchronisation auf das `this` Objekt ist „global“ und erlaubt auch nur einem einzigen Thread den Eintritt in diese Region (wie bei der Methodensynchronisation). Allerdings ist dieser Bereich deutlich kürzer als die Abarbeitung von `super.setRGB()` die bei der Methodensynchronisation ebenfalls innerhalb des globalen Sperrbereiches lag. Dies reduziert die Wahrscheinlichkeit, dass zwei Threads gleichzeitig diesen Lock versuchen zu erhalten, deutlich.

Die Ergebnisse Sprechen für sich. Auf Single-CPU Systemen skaliert diese Art des Lockings deutlich besser. Selbst mit 512 Threads ist keine messbarere Verlängerung der Berechnungszeit mehr zu verzeichnen. Auf 2 Prozessoren zeigt diese Art der Synchronisation einen ähnlichen Verlauf wie die Methodensynchronisation. Die benötigte CPU Zeit steigt bis 512 Threads um 18% und der Skalierungsfaktor Sinkt von knapp 2 auf 1.63. Offenbar ist die feine Synchronisation auf Single-CPU Systemen deutlich besser geeignet und führt auf Zweiprozessorsystemen zu keiner Verschlechterung.

7.4.3. CAS

Der letzte Test galt hier dem noch vorhandenen globalen Lock bei der Zähler-Inkrementierung. Diese wurde nun durch eine CAS (Lock-frei) ersetzt:

```
    public void setRGB(int startX, int startY, int w, int h,
                       int[] rgbArray, int offset, int scansize) {
        // increment counter using a CAS method.
        atomicCount.incrementAndGet();
        synchronized (locks[startX][startY]) {
            super.setRGB(startX, startY, w, h, rgbArray,
                         offset, scansize);
        }
    }
```

Listing 3 Code für CAS

Die Methode entspricht der Methode für feines Locking. Alleine der globale Lock wird nicht verwendet (`synchronized(this)`). An deren Stelle tritt die `incrementAndGet()` Methode einer `AtomicInteger` Klasse. Die Funktionsweise von CAS kann in [1] nachgelesen werden.

Wie erwartet entspricht das Ergebnis auf einer Single-CPU Maschine dem Ergebnis für feines Locking. Weder CPU- noch Berechnungszeit ändern sich in Grössenordnungen die nicht als Messrauschen bezeichnet werden können. Auf einer Mehrprozessor-Maschine trat aber eine Reduktion der CPU-Zeit und damit auch eine Reduktion der Berechnungszeit bei einer grossen Anzahl Threads ein. Bei 512 Threads stellte sich sowohl mit Methoden- als auch Objektsynchronisation auf unserer 2-CPU Plattform eine Erhöhung der CPU-Zeit und daran gekoppelt eine Erhöhung der Berechnungszeit um rund 20% ein. Mit CAS lag die Erhöhung der CPU-Zeit bei 512 Threads bei niedrigen 5%. In derselben Grössenordnung veränderte sich die Berechnungszeit.

CAS bietet hier also bei einer sehr hohen Anzahl Threads eine bessere Skalierung (rund 15% Vorteil).

Zu bemerken ist hier allerdings, dass die Länge der CAS Methode auch hier entscheidend für die Effizienz derselben ist. In unserem Beispiel besteht die Methode aus wenigen Codezeilen:

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Listing 4 CAS incrementAndGet() Methode

Eine Eigenschaft von CAS ist, dass der Algorithmus in einer Schleife durchlaufen wird. Dadurch wird der Thread nie blockiert. Allerdings kann der Thread die Methode nur beenden, wenn zwischen `get()` und `compareAndSet()` der Wert nicht durch einen anderen Thread verändert wird. Ansonsten muss die gesamte schleife erneut durchlaufen werden. Versuche mit unserer Testklasse haben gezeigt, dass dies bei unserer Architekture mit rund 700'000 Pixel-Schreibvorgängen nur wenige hundert Mal vorkommt. Der Zusatzaufwand durch den Ein- und Austritt bei synchronisierten Bereichen scheint hier massiv höher zu liegen als die wenigen fehlgeschlagenen `compareAndSet()` Aufrufe.

Sind die CAS Algorithmen aber länger, dann steigt die Wahrscheinlichkeit, dass zwischen `get()` und `compareAndSet()` der Wert durch einen anderen Thread verändert wurde. Dies führt zu häufigeren Neudurchläufen der der Schleife wobei natürlich je nach Komplexität auch entsprechend mehr CPU-Zeit reinvestiert werden muss.

In [4] wurden Vergleiche zwischen Java Synchronisierung und CAS durchgeführt. Dabei hat sich gezeigt, dass bei geringer bis mittlerer lock contention (konkurrierende Zugriffe) CAS besser skaliert. Bei sehr hoher lock contention ist aber die Synchronisierung effizienter. Dies hängt damit zusammen, dass bei hoher lock contention und vielen Threads viele Neudurchläufe der CAS-Funktion anfallen. Bei der Synchronisierung fällt dabei „nur“ der Aufwand für die Lock Anfrage und Lock-Abgabe an.

Bei unserem Beispiel ist der Aufwand für ein Neudurchlauf der `incrementAndGet()` Schleife deutlich geringer als die Lock-Prüfung beim fein granulierten Locking.

7.5. Empfehlungen

Als Faustregel sollten alle aufwändigen Berechnungen nach Möglichkeit auf mindestens so viele Threads wie verfügbare CPUs verteilt werden. Dadurch lässt sich im Optimalfall eine Skalierung um den Faktor n erreichen. Wobei n der Anzahl physikalischer CPUs entsprich. Je nach Häufigkeit und Dauer von eventuellen Blockierungen kann auch eine höhere Anzahl den Skalierungsfaktor erhöhen. Die Gesamtanzahl sollte aber nicht beliebig gross sein. Einerseits beeinträchtigt dies effizientes Scheduling und erhöht die Wahrscheinlichkeit für die gegenseitige Blockierung. Andererseits existiert auch ein Grenzwert bei dem in unseren Versuchen eine `OutOfMemoryException` auftrat (auch bei genügend freiem Speicher). Ein sauberes Applikations-Design mit einer definierten Anzahl von Threads (z.B. realisiert über Worker Pools) ist dringend zu empfehlen.

Der Programmierer muss darauf achten, dass sich die Threads nicht gegenseitig blockieren und zwar im Sinne von Deadlocks und Synchronisierung. Insbesondere sollte keine unnötige Synchronisation verwendet werden. Synchronisierte Blöcke sollten so kurz wie möglich gehalten werden.

CAS kann häufig dabei helfen den Synchronisierungsaufwand zu verringern. Da CAS Algorithmen aber schwerer zu implementieren sind wird in [4] sogar empfohlen dies den Experten zu überlassen. Allerdings kann durch die Verwendung von CAS-Methoden wie diejenigen der `Atomic*` Klassen CAS verwendet werden ohne den Algorithmus selber implementieren zu müssen. In unserem Beispiel hat sich der Einsatz von `AtomicInteger` mit einer 15% besseren Skalierung bezahlt gemacht und für eine beinahe lineare Skalierung bis 512 Threads gesorgt.

Die Objektsynchronisation stellt oft einen guten Kompromiss zwischen Methodensynchronisation und CAS dar. Der Entwickler sollte darauf achten, dass ein Lock-Objekt nach Möglichkeit für einen möglichst kleinen Teil der Daten verwendet wird (lock partitioning / lock striping). Insbesondere sollte vermieden werden ein Lock-Objekt für mehrere Datenobjekte, die nicht zusammen modifiziert werden, zu verwenden.

Vorsicht: Benötigt eine Methode den Lock auf zwei unterschiedliche Objekte (was durch lock partitioning nötig werden kann) besteht die Gefahr von Deadlocks! Beispiel:

```
public void doSomething() {
    synchronized(object1) {
        synchronized(object2) {
            object1.setValue(object2.getValue());
        }
    }
}
public void doSomethingElse() {
    synchronized(object2) {
        synchronized(object1) {
            System.out.println(obect1 + object 2);
        }
    }
}
```

Listing 5 Gefahr von Deadlocks bei verschachtelter Synchronisierung

Angenommen ein Thread ruft die Methode `doSomething()` auf und diese wird vor dem zweiten `synchronized()` Block unterbrochen (hält also den Lock auf `object1`. Ein zweiter Thread `ruft doSomethingElse()` auf, bekommt den Lock auf `object2` und wird dann ebenfalls unterbrochen weil er den Lock auf `object1` nicht bekommen kann (wird von einem anderen Thread gehalten). Nun benötigt jeder Thread eine Ressource, die nur ein anderer Thread freigeben könnte. Die Threads befinden sich in einem Deadlock.

7.6. Interpretation Testcase 10

JOMP bietet eine semi-automatische Parallelisierung an. Der Mandelbrot-Algorithmus lässt sich durch einfügen von JOMP Direktiven im Source-Code relativ einfach parallelisieren. Die Testergebnisse sind zunächst vielversprechend. JOMP erzeugt nach dem Setzen der korrekten Thread-Anzahl selbständig die benötigten Worker-Threads und verteilt die `for` Schleife in gleich grossen Teilen an die Worker. Die automatische Parallelisierung erzeugt keine messbar grössere CPU-Last als die manuelle Thread-Verwaltung.

Unsere Messungen haben wir ohne Locking durchgeführt um den Fokus auf die Effizienz der JOMP-Implementierung zu legen.

Die Ergebnisse sind vergleichbar mit den Messwerten der eigenen Thread-Behandlung und skaliert sowohl mit der Anzahl der Threads als auch auf 1 bzw. 2 CPUs in gleichem Masse wie die manuelle Implementierung.

Auf den ersten Blick scheint JOMP also eine gute Alternative zu sein bestehende Programme ohne Multi-Thread Architektur zu parallelisieren ohne grössere Re-Designs in Angriff nehmen zu müssen.

Die Praxis sieht aber leider etwas anders aus. Die JOMP Implementierung birgt (zumindest zum jetzigen Zeitpunkt) noch einige Tücken. Beispielsweise mussten wir feststellen, dass der JOMP-Compiler nur dann fehlerfreien Java-Code aus den JOMP-Klassen generiert wenn der Konstruktor alle Parameter in gleichnamige Klassenvariabeln schreibt. Werden die Parameter beispielsweise nur zur Erzeugung von klasseninternen Objekten verwendet lässt sich die von JOMP erzeugte Java-Klasse aufgrund von Kompilierungsfehlern nicht übersetzen.

Auch beim Design der Applikation müssen unter Umständen Änderungen vorgenommen werden. So mussten wir beispielsweise sicherstellen, dass nur eine einzige Instanz der JOMP-generierten Klasse aktiv ist. Bei mehreren Instanzen treten JOMP-interne Thread Verwaltungs-Fehler auf (Exceptions).

Beim Test ist uns dann aufgefallen, dass der Thread-Join am Ende der parallelen Region offenbar per Busy-Waiting realisiert wird. Somit bleibt die CPU-Last nach dem Ende der Berechnungen auf 100%. Diese Tatsache hat auch unsere CPU-Zeit Messungen etwas erschwert. Bei der Verwendung eines einzigen Threads tritt das „Problem“ nicht auf weil hier keine Verteilung und somit auch kein Join der Threads stattfindet.

All diese Probleme könnten auch durch den Beta Status (wir haben Version 1.0 Beta verwendet) erklärbar sein. Zum jetzigen Zeitpunkt können wir JOMP aufgrund der erwähnten Probleme nicht empfehlen. Insbesondere bietet die Java API bereits viele Möglichkeiten zur Parallelisierung wie beispielsweise Worker Pools, Queues, Locks, Barrieren. Und Latches. Trotzdem gibt JOMP einen guten Einblick in die OpenMP Programmierung und wir können uns gut vorstellen, dass OpenMP eine einfache und schnelle Art der Parallelisierung von C/C++ und Fortran Programmen anbietet.

8. Einsatzgebiete

Nicht alle Anwendungen bieten dieselben Optimierungsmöglichkeiten und lassen sich so einfach parallel verarbeiten wie unsere Mandelbrot Testklasse. Da stellt sich natürlich die Frage wo sich dieser Aufwand lohnt und wo nicht. Die folgende Tabelle teilt die Anwendungen grob nach Interaktivität und Berechnungsdauer auf:

Tabelle 5 Anforderungen an die Verarbeitungsgeschwindigkeit

	Interaktiv	**Nicht interaktiv**
Lange Berechnungsdauer	++	+
Kurze Berechnungsdauer	-	--

Unter den interaktiven Programmen sind dabei beispielsweise typische Desktop Anwendungen wie Office oder Bildbearbeitung zu verstehen. Zur Kategorie der nicht interaktiven Programme zählen im Hintergrund ablaufende Programme und Dienste.

Es liegt in der Natur des Benutzers, dass er nicht gerne auf die Maschine wartet. Deshalb ist eine schnelle Bearbeitung insbesondere bei interaktiven Programmen sehr wichtig um flüssiges Arbeiten zu ermöglichen. Für nicht-interaktive Programme ist dies weniger wichtig, da diese problemlos auch über längere Zeit laufen dürfen. Wichtig ist bei dieser Unterscheidung, dass einige auf den ersten Blick nicht-interaktive Anwendungen wie Webserver aus Sicht des Benutzers interaktive Programme darstellen, da der Benutzer auf das Resultat warten muss.

Trotzdem macht auch für nicht-interaktive Anwendungen eine Optimierung mit dem Fokus der parallelen Verarbeitung häufig sinn um die Effizienz der (teuren) Hardware zu verbessern.

Bei kurzer Berechnungsdauer rechtfertigt eine Optimierung zur parallelen Verarbeitung häufig nicht den Aufwand. Beispielsweise macht es selten Sinn die Initialisierung einer Dialogbox parallel zu verarbeiten. Hingegen macht es durchaus Sinn die Anwendung eines aufwändigen Filters einer Bildbearbeitung zu verteilen da der Benutzer üblicherweise auf das Ergebnis warten muss bis er weiter arbeiten kann.

9. Allgemeine Bemerkungen

Unsere Tests wurden natürlich auf einem Paradebeispiel für parallele Verarbeitung durchgeführt. Die Berechnung der Mandelbrot-Menge lässt sich ohne Reibungsverluste und völlig unabhängig auf mehrere CPUs oder gar auf mehrere Rechner verteilen. In der Praxis ist die leider nicht immer möglich. In den meisten Fällen können Aufgaben aber durch geschicktes Applikations-Design auf mehrere Threads verteilt werden. Im einfachsten Fall werden einfach die anfallenden Berechnungen nicht nacheinander sondern parallel verarbeitet. Dabei erschweren natürlich Abhängigkeiten diese Verteilung. Generell zeigen unsere Messungen aber, dass die parallele Verarbeitung weit mehr Leistung bringt als durch die Synchronisation wieder aufgehoben wird. Immerhin lässt sich durch die Verteilung aufwändiger Berechnungen auf 2 CPUs im Optimalfall nahezu die doppelte Leistung erzielen.

Ein weiterer, allgemeiner Pluspunkt für Multi-CPU/Multi-Core Systeme liegt darin, dass heutige Multitasking Betriebssysteme selten nur ein einziges Programm ausführen. So steht unter Umständen selbst für ein Single-Threaded Programm mehr CPU-Leistung zur Verfügung als auf einem Multi-CPU/Multi-Core System. Angenommen im Hintergrund läuft gerade die Festplatten-Defragmentierung oder ein Video-Encoder und im Vordergrund unsere Mandelbrot Berechnung mit nur einem Thread. In diesem Fall kann die Mandelbrot-Berechnung durch intelligentes Scheduling bis zu maximal 50% der zur Verfügung stehenden Ressourcen nutzen. Die restlichen 50% können dann von den anderen Programmen genutzt werden.

Der Alltag auf den meisten Desktop-Rechnern sieht aus der Sicht einer CPU aber eher langweilig aus. Die meiste Zeit verbringt der Prozessor dabei „schlafend“. Also warum braucht man dann gleich zwei oder gar noch mehr wartende Prozessorkerne? Die Antwort ist einfach: Für den Fall in dem die Leistung benötigt wird! Dabei zählen Microsoft Office Anwendungen sicher nicht zu diesen Killer-Applikationen. Diese begnügen sich unter Windows XP auch gerne mit einer 1GHz CPU. Häufig werden hier multimediale Anwendungen genannt doch wer hört schon gleichzeitig duzende MP3 Streams und sieht sich gleichzeitig einen Film in HDTV Qualität an? Selbst die Dekodierung eines MPEG2 Datenstroms zur DVD-Wiedergabe erledigt ein 600MHz Prozessor ohne Murren. Bei HDTV sieht die Welt dann schon wieder etwas anders aus. Hier werden wirklich schnelle Prozessoren benötigt. Man darf sich hier aber zu Recht fragen, ob ein einzelner, schnellerer Prozessor nicht besser wäre als mehrere (eventuell langsamere). Die Antwort darauf liegt in den bereits in der Basisanalyse ([1]). Die physikalischen Grenzen der Taktratenerhöhung treiben die Herstellungskosten in die Höhe. Somit ist es günstiger mehrere Kerne anzubieten als die einzelnen Kerne noch höher zu takten.

Um beim Beispiel HDTV-Decodierung zu bleiben würde bereits eine Trennung der Audio- und Video Decodierung eine gewisse Verteilung bringen. Da die Video-Decodierung dabei den Löwenanteil darstellt bietet sich eine Parallelisierung des Video-Algorithmus ebenfalls an. Dies ist aber je nach Komplexität des Algorithmus mit etwas Denksport verbunden.

Ähnlich sieht die Situation bei Spielen aus. Die meisten aktuellen Titel arbeiten immer noch Single-Threaded und laufen auf höher getakteten Single-Core CPUs somit schneller. Dies wird sich in absehbarer Zeit allerdings ändern. Neu angekündigte Titel werben bereits häufig mit dem Attribut „Multi-Core Unterstützung“ und meinen dabei meist die Auslagerung von KI-, Sound- oder Grafik-Berechnungen in eigenständige Threads. Das Attribut alleine sagt allerdings noch nichts über die Skalierung und die Effizienz der Verteilung aus und müsste individuell getestet werden.

Weitere Einsatzgebiete liegen natürlich im wissenschaftlichen Bereich. Anwendungen wie Maple, Mathematica oder Matlab erfordern starke Rechenwerke. Es bleibt aber im Einzelfall abzuklären, ob die favorisierte Anwendung auf Multi-CPU/Multi-Core Umgebungen optimiert ist.

10. Abschliessende Bemerkungen

Die Frage, ob man sich einen Multi-Core Prozessor kauft oder nicht wird sich in absehbarer Zeit gar nicht mehr stellen weil kaum mehr neue Single-Core Prozessoren auf den Markt kommen. Höchstens im Tiefpreis-Segment dürften diese noch einige Zeit zu bekommen sein.

Wie wir mit unseren Simulationen unter Last beweisen konnten bietet ein Multi-Core/Multi-CPU System auch bei ohne Optimierung (Single-Threaded) einige Vorteile. Insbesondere wenn mehrere Programme gleichzeitig CPU-Leistung benötigen. Dieser Fall hängt natürlich stark vom jeweiligen Anwenderprofil ab. Nicht unbedeutend kann dabei auch die vom Betriebssystem erzeugte Last sein. Windows XP führt beispielsweise bei geringer Systembelastung automatisch Optimierungen wie Indexaktualisierungen oder Defragmentierungen durch. Bei Maschinen mit mehreren Prozessoren sind solche Optimierungen auch während der Benutzung ohne wesentliche Beeinträchtigung möglich sofern ein Prozessor nicht bzw. kaum genutzt wird. Zu beachten ist dabei aber die Last auf gemeinsam genutzten Ressourcen. Insbesondere die Festplatte ist hier zu nennen. Eine Defragmentierung im Hintergrund könnte die Festplattenleistung merklich beeinträchtigen was wiederum (ungewollt) das Vordergrundprogramm beeinflussen kann.

Insgesamt kann man sagen, dass es nie verkehrt ist mehr Rechenleistung zum (fast) gleichen Preis zu bekommen. Allerdings sollte man die gebotene Rechenleistung nicht überbewerten und im Einzelfall abklären ob die verwendete Software davon profitiert. Im schlimmsten Fall kann die gewünschte Anwendung einen Dual-Core Prozessor nur zur hälfte ausnutzen. Hier sind also die Software-Hersteller gefragt. In dieser Arbeit haben wir einige Optimierungen und Technologien vorgestellt, die den Entwicklern mächtige Werkzeuge zur Verteilung der Aufgaben in die Hand geben. Ob diese genutzt werden liegt allerdings in den Händen der Entwickler.

Dass die vorgestellten Werkzeuge (insbesondere Java Threads unter Windows XP) sehr effizient arbeiten konnten wir eindrucksvoll nachweisen. Unsere Empfehlung lautet daher ganz klar die Optimierung rechenintensiver Aufgaben auf die parallele Verteilung. Optimierungen mittels Techniken wie TBB oder OpenMP/JOMP können sehr gut dazu beitragen bestehenden Code zu parallelisieren. Der Einsatz von POSIX/Win32/Java Threads bedeutet häufig etwas mehr strukturelle Änderungen und unter Umständen ein Re-Design der Anwendung bzw. Anwendungsteilen. Beim Design neuer Anwendungen sollte deshalb darauf geachtet werden, dass rechenintensive Teile parallel durch mehrere Recheneinheiten bearbeitet werden können. Nur so kann sichergestellt werden, dass die Bearbeitungszeit und somit die Wartezeit auf die Maschine so kurz wie möglich gehalten wird.

11. Glossar

Tabelle 6 Glossar

Begriff	Beschreibung
Affinität	Bezeichnet die Zuordnung eines Prozesses/Threads zu physikalischen Recheneinheiten. Durch die Definition einer Affinitätsmaske kann gesteuert werden auf welchen Recheneinheiten die Anwendung ausgeführt werden kann. Siehe Kapitel **Error! Reference source not found.**.
AMD	Advanced Micro Devices; Hersteller von Mikroprozessoren.
API	API (Application Programming Interface) defniert eine Schnittstelle zwischen verschiedenen Software Systemen. Eine API definiert typischerweise eine Reihe von Methoden, Parametern, Datentypen und Datenfeldern.
Berechnungszeit	Real vergangene Zeit, die ein Prozess/Thread benötigt um eine Aufgabe zu erledigen. Dies schliesst die gesamte Verarbeitungsdauer der Aufgabe ein und entspricht der Wartezeit, die der Benutzer auf ein Ergebnis warten muss. Vergleiche auch mit „CPU-Zeit“.
CAS	Compare and Swap bzw. Compare and Set bezeichnet eine atomare (meist hardwareunterstützte) Operation in der ein gespeicherter Wert mit dem vermuteten Wert verglichen wird. Stimmt dieser überein, so wird ein neuer Wert gesetzt. Ansonsten wird nichts getan. CAS Funktionen erlauben Lock-freie Algorithmen. Siehe Kapitel 7.4.3.
CPU	Abkürzung für Central Processing Unit. Wird synonym für die deutsche Bezeichnung Hauptrpozessor bzw. Prozessor verwendet.
CPU-Zeit	Die Gesamtzeit, die von einem Prozess/Thread für die Ausführung in Anspruch genommen wird. Also die Gesamtzeit in der ein Prozess/Thread auf der Hardware ausgeführt wird. Hinweis: Bei einem 2-Prozessor-System kann ein Programm mit einer Laufzeit von 10 Sekunden durchaus 20 Sekunden CPU-Zeit „verbrauchen“ da die 10 Sekunden über zwei Threads auf beiden CPUs belegt werden.
GUI	Graphical User Interface; Bezeichnet die Darstellung der Benutzeroberfläche durch grafische Elemente wie Knöpfe, Symbole, Menüs und Zeichnungen.
Hyper-Threading	Eine von Intel bei einigen Pentium 4 Modellen eingeführte Technologie zur verbesserten Auslastung der internen Pipeline. HyperThreading stellt gegenüber dem Betriebssystem einen zweiten (virtuellen) Prozessor zur Verfügung. Dieser ist aber physikalisch gar nicht vorhanden. Instruktionen an diesen Prozessor können die Auslastung der internen Rechen-Einheiten des Pentium 4 verbessern.
JOMP	Java-basierende Implementierung von OpenMP-Ähnlichen Direktiven zur Parallelisierung. Siehe Kapitel 7.6.
JVM	Die Java Virtual Machine ist ein Interpreter für Java Bytecode. Die JVM ist dabei das Bindeglied zwischen Betriebssystem und den plattformunabhängigen Java Anwendungen.
NUMA	Non-Uniform Memory Access (NUMA) bezeichnet eine Architektur in der jede Verarbeitungseinheit lokalen Speicher besitzt und durch Kommunikation mit den anderen Verarbeitungseinheiten auch deren Speicher ansprechen kann.

Begriff	Beschreibung
Scheduling	Bezeichnet die Tätigkeit des Betriebssystems beim Preemptiven Multitasking die Prozessorzeit nach einem bestimmten Algorithmus den einzelnen Ausführungseinheiten zuzuweisen (auf Ebene Thread oder Prozess).
TBB	Intel Thread Building Blocks. Eine C++ Bibliothek die Methoden zur parallelen Verarbeitung bereitstellt (Schleifenparalleisierung).
Thread	Ein leichtgewichtiger Prozess. Ein Thread teilt den Adressraum mit dem Prozess zu dem er gehört. Dadurch werden einerseits die Kommunikation und andererseits der Kontextwechsel beschleunigt. Siehe Kapitel 7.

12. Verzeichnisse

12.1. Tabellenverzeichnis

Tabelle 1 Referenzierte Dokumente 5
Tabelle 2 Abkürzungen 5
Tabelle 3 Links 5
Tabelle 4 Bereich der Basisprioritäten in Abhängigkeit der Prozessprioritätsklasse 7
Tabelle 5 Anforderungen an die Verarbeitungsgeschwindigkeit 20
Tabelle 6 Glossar 23

12.2. Abbildungsverzeichnis

Abbildung 1 Verteilung einer Single-Thread Anwendung auf zwei CPUs 10
Abbildung 2 Verteilung von zwei Threads auf zwei CPUs 10
Abbildung 3 2 Threads mit gesetzter CPU Affinität auf CPU0, ohne Last 10
Abbildung 4 Skalierung 1 CPU/2 CPU mit Methodensynchronisation 14
Abbildung 5 Thread-Abarbeitung 15

12.3. Code Listings

Listing 1 Methodensynchronisation 14
Listing 2 Code für feines Locking (Objektsynchronisation) 16
Listing 3 Code für CAS 16
Listing 4 CAS incrementAndGet() Methode 17
Listing 5 Gefahr von Deadlocks bei verschachtelter Synchronisierung 18

12.4. Index

Abkürzungen 5
Affinität 10, 23
AMD 23
API 23
Basispriorität 7
Berechnungszeit 23
CAS 16, 18, 23
CPU 23
CPU-Zeit 23
Definitionen 5
Effizienz 20
Einsatzgebiete 20
GUI 12, 23
HDTV 21
Hyper-Threading 23
JOMP 19, 22, 23
JVM 23
Links 5
Methodensynchronisation 14
Multitasking 21
NUMA 11, 13, 23
OpenMP 19, 22
POSIX 22
Prozesspriorität 7
Referenzen 5
Round-Robin 15
Scheduling 8, 24
TBB 22, 24
Thread 24