

Synchronisation von Threads zur Optimierung des Rechnens auf Mehrkernprozessoren am Beispiel der Simulation von Planetenbewegungen

Markus Pargmann (363/05)

10. Dezember 2007

Inhaltsverzeichnis

1	Einleitung	3
2	Multicore-Architekturen	4
3	Scheduler	4
3.1	Prozess-Modell	4
3.2	Thread-Modell	5
4	Threads	5
4.1	Funktionsweise	5
4.2	Problematik	6
4.2.1	Determinismus	6
4.2.2	Synchronisation	6
4.2.3	Probleme bei der Synchronisation	7
4.3	Optimierung	8
5	Threads in Java	8
5.1	sleep	9
5.2	yield	9
6	Synchronisation der Threads	9
6.1	Ringstruktur von Threads	10
6.1.1	Performancevorteil	10
6.1.2	Problem des Determinismus	10
6.2	Synchronisation über sleep	10
6.2.1	Geschwindigkeitsverlust	11
6.3	Synchronisation über yield	11
6.3.1	Geschwindigkeitsvorteil	11
6.4	Nachteil	11
6.4.1	Aufwand des Programms	12
6.4.2	Grund der schlechten Performance	12
7	Fazit	14
8	Literaturverzeichnis	15
9	Anhang	16
9.1	Implementierung in Java	16
9.2	Optimierte Klasse Calculation	17

1 Einleitung

In dieser Facharbeit möchte ich auf ein aktuelles Problem eingehen, das sich auf neue Prozessoren bezieht. Seit etwa einem Jahr gibt es nun die Prozessoren, die mehrere Prozessorkerne besitzen. Dabei sind in einem Prozessor mehrere Prozessoren vereinigt, die dafür sorgen, dass mehrere Prozesse wirklich parallel bearbeitet werden können und nicht nur anscheinend parallel laufen, wie es ein Scheduler ermöglicht. Dieser weist verschiedenen Prozessen die CPU zu.

Durch diese neue Art von Prozessoren müssen folglich auch Programme anders konzipiert werden, da sie nun mehrere Kerne nutzen können. Benutzt ein Programm nur einen Thread zur Ausführung, so bringen diesem Programm mehrere Prozessoren oder Prozessorkerne keinen Vorteil. Ein Prozess kann nicht ohne Analyse der Befehle in mehrere parallele Prozesse aufgeteilt werden, da gewisse Abhängigkeiten zwischen Befehlen bestehen. Deshalb müssen die Programme an diese Mehrkern-Architekturen angepasst werden. Dazu ist es notwendig ein Programm in mehrere Prozesse aufzuteilen, die dann auf den verschiedenen Prozessoren laufen und somit die Ausführungsgeschwindigkeit des Programms steigern können.

Dabei muss jedoch beachtet werden, dass die einzelnen Prozesse nicht mehr gezwungen sind in der gleichen zeitlichen Reihenfolge zu laufen. Das bedeutet, dass Prozesse vollkommen unabhängig voneinander sein müssen, um zu gewährleisten, dass das Resultat sich nicht bei jeder Programmausführung ändert oder das ganze Programm nicht mehr funktioniert. Um die einzelnen Prozesse trotzdem in das Programm einzubinden, ist die Synchronisation der Prozesse sinnvoll. Unter anderem wird so auch erreicht, dass Daten ausgetauscht werden können.

Wie stark man optimieren kann wird vor allem bei rechenintensiven Anwendungen deutlich, bei denen es viele unabhängige Rechnungen gibt, da hier durch das Ausnutzen mehrerer Prozessoren ein erheblicher Geschwindigkeitsvorteil erreicht werden kann. Hier werde ich auf die Grundlagen eingehen die man benötigt um möglichst optimal Programme für mehrere Prozessoren zu schreiben. Anschließend werde ich auf die Optimierung von Programmen eingehen, die keine langen unabhängigen Rechnungen haben. Da dies oft bei Simulationen der Fall ist, werde ich die Optimierung an einer selbstimplementierten Simulation von Planetenbewegungen in Java darstellen. Hierbei handelt es sich um die Simulation von mehreren punktförmigen Massen unter Berücksichtigung der Gravitation und Vernachlässigung von Kollisionen.

Die Frage die sich hier stellt ist, wie man eine solche Simulation gut in Prozesse aufteilen kann und diese dabei möglichst optimal synchronisiert. In dieser Facharbeit werde ich auf ein mögliches Verfahren im Vergleich zu anderen Synchronisationsmöglichkeiten eingehen. Die Realisierung von Threads in anderen Programmiersprachen außer Java werde ich nicht ansprechen.

Meine Facharbeit behandelt nur die Optimierung von Programmen am Beispiel von Mehrkernprozessoren. Die Probleme und Lösungen können teilweise auf andere Multicore-Architekturen übertragen werden.

2 Multicore-Architekturen

Es gibt verschieden Arten wie Multicore-Architekturen aufgebaut sein können. Es gibt die Möglichkeit, dass mehrere Prozessoren über Motherboards oder Netzwerke verbunden sind. Solch eine Architektur hat das Problem, dass die Bandbreite der Verbindung recht gering sein kann und die Leitungen zum anderen Prozessor relativ weit sind, sodass eine Kommunikation sehr langsam ist.

Eine weitere Architektur, die mehrere Prozessoren zur Verfügung stellt, ist wie ein Gitter aufgebaut, wobei jeder Kern nur mit seinen vier Nachbarn kommunizieren kann. Dies ist eine sehr außergewöhnliche Architektur und unterstützt keine der bekanntesten Befehlssätze, wie Intels i386, Apples PowerPC oder die Befehlssätze von Sun.

Bei den im Moment am meisten verbreitetsten Multicore-Architekturen von Intel und AMD sind in einem Prozessor mehrere Prozessorkerne vereint. Sie verstehen alle den alten i386 Befehlssatz. Dabei gibt es bei diesen Architekturen Unterschiede in der Realisierung. So gibt es bei manchen Prozessoren Caches, die von mehreren Prozessorkernen benutzt werden. Bei gemeinsam verwendeten Caches gibt es den Vorteil, dass gemeinsam benutzte Speicheradressen nur einmal im Cache liegen müssen. Bei diesem Konzept von gemeinsamen Caches sind im Moment nur die Level 2 und 3 Caches betroffen. Bislang macht es keinen Sinn den Level 1 Cache mit mehreren Kernen gemeinsam zu nutzen, da hier nur die möglichen nächsten Befehle gespeichert sind und teilweise ihre Operanden.

Diese Caches werden stark genutzt um die langen Zugriffszeiten auf den Hauptspeicher zu vermeiden. Das Problem der hohen Latenz tritt auf, da der Takt des Prozessors mittlerweile sehr viel höher ist als der des Speichercontrollers, der für die Speicheranbindung zuständig ist. Somit muss der Prozessor immer auf die Daten vom Speicher warten. Um dies zu vermeiden wird mit dem Level 1 Cache versucht möglichst viele Befehle, die im weiteren Programmablauf folgen können, schon vorher aus dem Speicher zu laden. So hat die CPU den Befehl vorliegen sobald sie ihn benötigt. Dabei ist es nicht möglich alles korrekt im Voraus zu laden, da man für die korrekte Interpretation der Befehle eine weitere CPU benötigt, bei der dann jedoch auch wieder das Laden der Befehle zu lange dauern würde. Somit wird nur „geschätzt“ welchen Befehl der Prozessor als nächstes benötigt. Im Level 2 und Level 3 Cache werden häufig benutzte Speicherblöcke zwischengespeichert. Zum Beispiel schreibt die CPU auf den Cache anstatt auf den Speicher. Die Daten werden dann, während die CPU schon den nächsten Befehl verarbeitet, mit dem Speicher abgeglichen. So wird das Problem des langsamen Speichers, was auch als Von-Neumann-Flaschenhals bekannt ist, etwas umgangen.

3 Scheduler

3.1 Prozess-Modell

Scheduler sind ein Bestandteil von Betriebssystemen. Diese sind dafür zuständig die verschiedenen Prozesse, die in einem Betriebssystem ablaufen, zu managen. Verschiedene Prozesse können dabei verschiedene Prioritäten bekommen. Unter Berücksichtigung dieser Prioritäten bekommen die Prozesse nacheinander eine bestimmte Zeit, in der sie auf

der CPU laufen dürfen. Nach einer bestimmten Zeit wird vom Prozessor ein Interrupt erzeugt und dadurch eine Interruptbehandlung gestartet. Diese sorgt dafür, dass die Werte der Register, des Programcounters und des Stack gespeichert werden. Danach wird entschieden welchen Prozess die CPU als nächstes benutzen darf. Dementsprechend werden die Werte des nächsten Prozesses geladen. Des Weiteren wird der verfügbare Speicherbereich geändert um zu verhindern, dass verschiedene Prozesse die Möglichkeit haben auf den gleichen Speicherbereich zuzugreifen. Benötigt ein Prozess weniger Zeit auf dem Prozessor als er bekommen hat, so bekommt er beim nächsten mal weniger Zeit auf der CPU, sonst mehr. Wartet ein Prozess auf IO-Aktionen (Input-/Output-Aktionen), so wird er von der CPU schneller verdrängt, weil er nur wartet. Außerdem können Prozesse durch ankommende IO-Aktionen unterbrochen oder verdrängt werden um die schnelle Bearbeitung dieser Daten zu ermöglichen.

3.2 Thread-Modell

Das Thread-Modell ist ähnlich wie das oben beschriebene Prozess-Modell. Hierbei ist jedoch jeder Prozess unterteilt in sogenannte Threads. Diese kann man sich als Ausführungsfaden vorstellen. Jeder Prozess beinhaltet dann mehrere dieser Threads. Diese benutzen den gleichen Speicherbereich des Prozesses zu dem sie gehören. Über den Speicher ist es den Threads dann möglich zu kommunizieren. Je nach Schedulerimplementierung ist es auch möglich die Register oder Stapelspeicher eines anderen Prozesses auszulesen.

Dieses Thread-Modell hat den Vorteil, dass die Threads aus einem Prozess, der geeignet implementiert sein muss, auf mehreren Prozessorkernen parallel laufen können und somit die Last verteilt wird. Ein weiterer Vorteil ist, dass der Wechsel zwischen zwei Threads in einem Prozess auf einem Prozessor schneller geht, als ein Wechsel zwischen zwei Prozessen, da bei allen Threads aus einem Prozess der zugewiesene Speicherbereich der Gleiche ist und somit nur die Register und Stacks neu geladen werden müssen.

Aus diesen Gründen gibt es in modernen Betriebssystemen fast nur noch Scheduler, die ein Thread-Modell implementieren.

4 Threads

4.1 Funktionsweise

Wie oben bereits erläutert gibt es in einem Prozess mindestens einen Thread. Dieser wird mit Beginn des Prozesses gestartet. Weitere Threads können über schon bestehende Threads gestartet werden. Bei dem Starten eines Threads wird ein Zeiger übergeben. Dieser zeigt auf eine Funktion, welche in einem neuen Thread aufgerufen wird und wie der Hauptthread funktioniert. Das bedeutet, solange nicht jeder Thread beendet ist, ist auch der Prozess nicht beendet. In dieser aufgerufenen Funktion können erneut neue Threads erzeugt werden.

Beendet werden kann ein Thread durch sich selbst oder einen anderen Thread aus dem gleichen Prozess. Dabei gibt es zwei unterschiedliche Möglichkeiten einen Thread zu beenden. Zum einen kann der Thread zusammen mit allen Daten, die er in den

Registern gespeichert hat, beendet und die Register und Stacks sofort gelöscht werden. Zum anderen gibt es die Möglichkeit, dass die Register des Threads gespeichert werden, bis ein Thread die Informationen erfragt hat. Erst danach werden sie gelöscht. Bei beiden Varianten bleiben die Informationen im Speicherbereich erhalten, da diese nicht Thread-spezifisch sind, sondern zu dem gesamten Prozess gehören.

Zusätzlich gibt es für Threads die Möglichkeit, dass sie ihre Zeit auf der CPU sofort beenden. Das bedeutet, dass der Thread dann sofort von der CPU genommen wird. Die verbleibende Zeit wird an einen anderen Thread gegeben. Diese Variante ist sehr wichtig, da es sehr viele Programme gibt, die etwas ständig überprüfen müssen, zum Beispiel irgendwelche Sensoren, um auf Änderungen reagieren zu können. Bei den aktuellen Taktraten ist es vollkommen unnötig während der Überprüfung keine Pausen einzulegen, da sich in der Zeit meist nichts ändert, oder eine verspätete Reaktion des Prozesses auf die Änderung nicht auffällt. Daher kann man in einer solchen Situation diese Möglichkeit nutzen und zum Beispiel einmal den Sensor überprüfen und schließlich die CPU wieder an einen anderen Thread oder einen anderen Prozess abgeben, die die Zeit anders und sinnvoller nutzen können.

4.2 Problematik

4.2.1 Determinismus

Es gibt einen großen Unterschied zwischen dem Programmieren mit nur einem und mehreren Threads. Programmiert man mit nur einem Thread, so ist einem unmittelbar klar, dass Daten, die man benutzt, stets die aktuellsten sind. Es gibt schließlich nur einen Befehlsstrang, der auf den Speicherbereich Zugriff hat. Somit hat man einen klaren linearen Programmablauf, bei dem man mit Sicherheit wissen kann, was in einer Variablen oder im Speicher steht.

Bei der Programmierung mit mehreren Threads bekommt man hier Probleme. Da der Scheduler den Threads unabhängig voneinander den Prozessor zuweist, kann man während des Programmierens nicht wissen, welcher Thread vor welchem Thread die CPU bekommt, der lineare Programmablauf existiert nicht mehr. Wenn mehrere Threads eine Variable beschreiben, und ein Thread diese auch liest, so kann es schnell dazu kommen, dass die Variable bei jedem Lesen einen anderen Wert hat. So ist der Ablauf des Prozesses ungewiss und kann bei jeder Ausführung anders sein. Dies passt zum einen nicht mit der Definition eines Algorithmus zusammen, die vorsieht, dass bei gleichen Eingabeparametern immer das Gleiche herauskommen muss (Determinismus), zum anderen ist es für die meisten Prozesse nicht sinnvoll. Deshalb sollte man bei der Programmierung mit Threads darauf achten, dass sich die Threads bei Zugriff auf die gleiche Speicheradresse vorher „absprechen“, also sich synchronisieren. Dadurch soll verhindert werden, dass Threads die falschen Werte einlesen.

4.2.2 Synchronisation

Durch die Synchronisation erreicht man, dass Programme mit mehreren Threads, die die gleichen Variablen benutzen, wirklich deterministisch bleiben. Um zwei oder mehr

Threads zu synchronisieren, müssen diese miteinander kommunizieren und ihren Programmstand austauschen. Dies kann im einfachsten Fall bedeuten, dass ein Thread bei der Kommunikation zwei Zustände angeben kann: „fertig“ oder „am arbeiten“. Diese Zustände können aber beliebig erweitert werden.

Die Kommunikation zwischen zwei Threads ist aber leider nicht so einfach wie es sich anhört. Da die CPU keine Möglichkeiten anbietet, womit zwei Threads kommunizieren können, muss auf die einzige gemeinsame Resource zurückgegriffen werden, wo Daten über längere Zeit abgespeichert werden können, also der Speicher. Somit muss der wartende Thread zum einen seinen eigenen Zustand im Speicher ablegen, zum anderen muss er auf den Speicher zugreifen um den Zustand der anderen Threads zu erfahren. Also greift er in gewissen Abständen immer wieder auf den Speicher zu um zu kontrollieren ob der andere Thread bereits fertig ist.

4.2.3 Probleme bei der Synchronisation

Wenn zwei Threads die gleichen Unterprogramme aufrufen, kann es zu Problemen führen, da der eine Thread verdrängt werden könnte und die Funktion noch nicht abgearbeitet ist. Das wird problematisch, da dann beide Threads die gleichen Speicheradressen, die in der Funktion angegeben sind, nutzen könnten. Um dies zu verhindern gibt es mehrere Varianten. Zwei werde ich im Folgenden kurz vorstellen, zum einen auf Hardwareebene, zum anderen auf Betriebssystemebene.

Die CPU stellt für solch ein Problem die Möglichkeit zur Verfügung, keine Unterbrechungen durch Interrupts zuzulassen. So arbeitet der Thread ohne eine Unterbrechung, also in Echtzeit. Das hat den Vorteil, dass die Ausführung auf jedenfall bis zu einem bestimmten Punkt gemacht wird. Dafür gibt es in der CPU eine sogenannte Flag, die man setzen muss um diese Option zu aktivieren. Sobald die Flag gesetzt ist, wird das laufende Programm nicht mehr unterbrochen. Wird sie wieder zurückgesetzt, können sofort Interrupts den Programmablauf unterbrechen. Beim Nutzen dieser Option muss jedoch beachtet werden, dass bei langen Ausführungen ohne Unterbrechung ein Interruptstau auftreten kann. Im schlimmsten Fall ist der Stau so lang, dass Interrupts verfallen und nichtmehr bearbeitet werden. Außerdem löst dies bei den neuen Prozessoren mit mehreren Kernen nicht mehr die Problematik, da bei mehreren Prozessorkernen die Threads immer noch gleichzeitig den Speicher nutzen und somit Überschneidungen auftreten können.

Auf Softwareebene ist es möglich eine Methode als blockiert zu kennzeichnen. Dabei wird eine einfache Variable gesetzt, wenn die Funktion blockiert ist. Das bedeutet, dass die Funktion bereits von einem anderen Thread benutzt wird. Dadurch wird verhindert, dass mehrere Threads gleichzeitig die Methode und damit die gleichen Speicheradressen nutzen. Threads die darauf warten, dass diese Funktion wieder verfügbar ist, werden in dieser Zeit als blockiert gekennzeichnet und bekommen keine Zeit auf dem Prozessor zugeteilt. Solch einen Status bekommen auch Threads, die darauf warten, dass sie Ressourcen, wie zum Beispiel die Festplatte, wieder nutzen können. Aber auch bei dieser Lösung können Probleme auftreten. In seltenen Fällen kann es passieren, dass sich mehrere Threads gegenseitig blockieren. Auf solche Eventualitäten muss in der Software

eingegangen werden, da es sonst möglich ist, dass das komplette Programm anfängt in einer Art blockiertem Zustand zu sein. Denn durch Synchronisation können die anderen Threads auf die blockierten Threads warten ohne das etwas passiert. Sind die blockierten Threads unabhängig, so arbeiten nur diese nicht mehr.

Es gibt einige andere Probleme die noch auftreten können, auf die ich hier jedoch nicht weiter eingehen möchte.

4.3 Optimierung

Um ein Programm durch Threads zu optimieren muss man sich erst einmal überlegen, ob das Programm überhaupt auf vielen Mehrkernprozessoren zum Einsatz kommen wird. Ist dies nicht der Fall, so erreicht man durch mehrere Threads nur, dass das Programm langsamer wird, da die CPU häufiger zwischen Threads umschalten muss. Des Weiteren wird die unnötige Kommunikation zwischen Threads gemacht. Außerdem ist diese Optimierung auch nur bei Programmen nötig, die Rechenintensiv sind. Der extra vorgenommene Programmieraufwand ist sonst ungerechtfertigt, da das Programm gar nicht an die Grenze der Leistungsfähigkeit des Prozessors stößt.

Bei rechenintensiven Programmen ist der Einsatz von Threads sinnvoll. Hier kann man unabhängige Codeabschnitte gut in einzelnen Threads unterbringen. Dabei ist es wichtig, dass die Codeabschnitte, die am meisten Prozessorlast erzeugen, in unterschiedlichen Threads sind. So kann man mehr als einen Prozessor beschäftigen. Des Weiteren sollte man auf möglichst geringe Speichernutzung achten, weil alle beteiligten Prozessoren denselben Speicher nutzen müssen. Es kann hier nicht auf das neue Dual-Channel System des Speichers zurückgegriffen werden. Das moderne Dual-Channel System erlaubt es, zwei verschiedene Speicherriegel über verschiedene Prozessoren gleichzeitig anzusprechen. Das kann hier nicht benutzt werden, weil der gesamte Prozess einen Speicherbereich hat und somit beide Prozessoren auf diesen einen Speicherbereich zugreifen müssen, wenn die Threads des Prozesses gleichzeitig auf den verschiedenen Prozessoren laufen. Damit kommt es dazu, dass die Prozessoren teilweise darauf warten müssen, dass die Anfrage des anderen Prozessors an den Speicher beendet ist. Also ist es bei Programmen, die mehrere Threads nutzen, noch wichtiger möglichst wenig Aktionen auf dem Speicher auszuführen.

Es gibt aber auch eine andere gute Nutzung von Threads, die nicht die Optimierung von Programmen sondern die praktische Nutzung von Threads zum Ziel hat. Als Beispiel kann man Threads nutzen um parallel neben dem Ablauf des Programms auf Eingaben durch den Benutzer zu warten und zu reagieren. So können Eingaben von Benutzern schnell verarbeitet werden und gleichzeitig wird der Programmfluss nicht unterbrochen.

5 Threads in Java

In Java hat man über die Klasse Thread die Möglichkeit Threads zu erstellen. Hat man ein Objekt der Klasse Thread, so kann der Thread durch die Methode `start()` gestartet werden. Bei dem Aufruf wird in dem Objekt die Methode `run()` aufgerufen. Ist diese Methode abgearbeitet, so ist der Thread beendet. Das Objekt bleibt bestehen, aber

fungiert nicht mehr als Thread und kann auch nicht erneut gestartet werden. Es gibt zwei Möglichkeiten die Funktion zu definieren, die der Thread ausführen soll.

Zum einen kann man das Interface Runnable implementieren. Dort muss man die Methode run() implementieren. Diese wird dann im Thread ausgeführt. Von dieser Klasse Runnable muss ein Objekt an den Konstruktor von der Klasse Thread übergeben werden. Bei Aufruf von start() wird dann die Methode run() vom Objekt des Typs Runnable ausgeführt.

Zum anderen kann man eine eigene Klasse erstellen, die von der Klasse Thread erbt. Auf Objekten dieser eigenen Klasse kann dann auch die Methode start() ausgeführt werden, die die Methode run() in einem Thread aufruft.

5.1 sleep

Diese Methode existiert in der Klasse Thread und sorgt dafür, dass der Thread eine gewisse Zeit lang nicht läuft. Angegeben werden kann die Zeit in Millisekunden und Nanosekunden. Während der angegebenen Zeit wird der Thread nicht ausgeführt. Das bedeutet er produziert keinerlei Last auf dem Prozessor. Solch eine Methode kann man gut nutzen, wenn zum Beispiel bestimmte Variablen in Zeitintervallen kontrolliert werden müssen. Zwar ist der Thread während dieser Zeit am „schlafen“, jedoch ist die Verzögerung in der diese Änderungen erkannt werden für viele Anwendungsgebiete akzeptabel. Dadurch wird anderen Threads ermöglicht die CPU sehr viel stärker zu belasten.

5.2 yield

Diese Methode implementiert die Möglichkeit, dass ein Thread die CPU verlässt, um sie einem anderen Thread zu überlassen. Auch diese kommt aus der Klasse Thread. Sie gewährleistet eine schnelle Umschaltung zwischen Threads. Sie ist sehr nützlich wenn das Programm noch zu rechnen hat, aber ein anderer Thread zuerst behandelt werden soll.

6 Synchronisation der Threads

Im Folgenden werde ich drei verschiedene Verfahren der Synchronisation für Simulationen vorstellen, die ich für die Simulation von Planetenbewegungen entwickelt habe. Mein Ziel war es, die Simulation deterministisch zu gestalten und dabei eine deutliche Geschwindigkeitssteigerung im Vergleich zur linearen Programmierung zu erzielen.

Das Programm was darum existiert ist sehr einfach gehalten. Es gibt ein Hauptthread, der bei Aufruf des Programms (java Umlaufbahn) aufgerufen wird. Dort werden die Übergabeparameter umgesetzt, zum Beispiel die Auflösung des Bildschirms, die Positionen der Planeten, etc. Dieser Hauptthread erstellt dann ein „Universe“ in dem alle Eigenschaften der Berechnung gespeichert sind, sowie alle Planeten und Rechnungen der Planeten. Des Weiteren wird dort ein Objekt für die grafische Darstellung erzeugt. Dies ist ein Thread und zeichnet in regelmäßigen Abständen die Planeten auf den Bildschirm.

Jeder Planet, der nicht an einer Position fixiert ist, hat einen eigenen Thread der für das Rechnen zuständig ist (Calculation). Außerdem existiert eine Liste, die alle Planeten enthält. In den Objekten die rechnen, also den Calculation Objekten, ist diese Liste bekannt. Aus dieser werden die Daten genommen, die gebraucht werden um die Gravitation zwischen zwei Planeten zu berechnen. Später werden die Daten in der Liste mit Planeten wieder überschrieben. Da jedes Objekt des Typs Calculation jeden Planeten zur Berechnung braucht, ist es wichtig, dass die Daten aktuell sind. Daher braucht man hier eine Synchronisation.

6.1 Ringstruktur von Threads

In diesem Ansatz habe ich die Calculation Objekte in einem Ring angeordnet. Jedes Objekt kennt seine beiden Nachbarn. In jedem Calculation Objekt habe ich einen Counter eingeführt. Dieser zählt die Anzahl der ausgeführten Rechnungen mit. Bei jeder Rechnung hat das Objekt seinen Zählerstand mit denen seiner Nachbarn abgeglichen. Ist dieser um einen bestimmten Wert höher als die der Anderen, so pausiert sich der Thread selbst. Ist der Wert nicht höher, so werden die Nachbarobjekte wieder gestartet. So wird garantiert, dass kein Thread weiter gerechnet hat als seine Nachbarn, ohne Beachtung der Toleranzgrenze, die bestimmt, ab wie viel Differenz gestoppt werden soll.

6.1.1 Performancevorteil

Durch die Tolleranzgrenze ist es möglich, dass Threads über längere Zeit ohne zu stoppen arbeiten können. Durch das Überprüfen des Zählerstands der Nachbarobjekte in Intervallen kann auch die Benutzung des Speichers reduziert werden. Das bedeutet eine weitere Steigerung der Performance. Durch diese Anordnung und Ausführung ist es eine relativ effiziente Nutzung des Prozessors, da die Synchronisation nicht bei jeder Rechnung gemacht werden muss.

6.1.2 Problem des Determinismus

Leider erfüllt dieser Ansatz nicht die Vorgaben, dass das Programm deterministisch sein soll. Durch den Tolleranzwert, der angibt wie viele Rechnungen ein Thread einem seiner Nachbarn voraus sein darf, passiert es, dass ein Thread mehrere Schritte nacheinander macht. Dabei bleiben teilweise die Werte der anderen Planeten konstant. Also wird hier mit Werten gerechnet, die sich nach einem Rechenschritt bereits wieder geändert haben sollten. Die Werte ändern sich aber nur für die Planeten deren zugehörige Calculation Objekte gerade auf einem Prozessor berechnet werden. Alle anderen Planetendaten ändern sich in diesem Zeitraum nicht.

6.2 Synchronisation über sleep

In diesem Ansatz habe ich die Idee, die Calculation Objekte in einem Ring anzuordnen, übernommen. Nun werden die Nachbarn jedoch nicht mehr verglichen, da dies offensichtlich zu Abweichungen führt. Somit fallen einige Variablen weg. Da es keine Vergleiche

mehr gibt, ist auch die Nutzung des Speichers für die Synchronisation zurückgegangen. Das bedeutet das Rechnen wird schneller. Leider muss eine Synchronisation vorhanden sein um das Ziel zu erreichen, deshalb habe ich einen weiteren Thread angelegt, der rekursiv in dieser Ringstruktur synchronisiert. Die Synchronisation ist in zwei Abschnitte aufgeteilt. Der erste Abschnitt umfasst das Laden der benötigten Variablen und das Ausrechnen der neuen Daten. Im Zweiten werden dann die errechneten Daten der Planeten gespeichert. Die Ringstruktur hat Methoden, die diese beiden Abschnitte einleiten. Das Objekt, welches für die Synchronisation zuständig ist, ruft dann eine dieser Methoden auf. Dieses Objekt ist ein weiterer Thread und übergibt eine Referenz auf sich selbst an die Methoden, um sich selbst schlafen zu lassen und damit den Prozessor zu schonen und einem anderen Thread die Möglichkeit zu geben zu rechnen.

6.2.1 Geschwindigkeitsverlust

Dadurch, dass die Threads nicht mehr längere Zeit an einem Stück laufen können und öfter aufhören müssen wird die Berechnung insgesamt langsamer. Außerdem wird die Ausführung dadurch gebremst, dass die Threads immer ein vielfaches der in sleep angegebenen Zeit warten müssen. Somit warten sie oftmals länger als nötig ist. Das summiert sich bei den vielen Synchronisationsschritten die gemacht werden müssen, sodass diese Zeit durchaus spürbar wird.

6.3 Synchronisation über yield

In diesem Ansatz verwende ich weiterhin die Ringstruktur der Rechnungen. Auch das Prinzip des Synchronisationsobjekts und die Funktionsweise bleiben wie oben beschrieben. Ich setze hier nur an dem Problem an, welches die zu hohen Zeiten waren, die ein Thread schlafen kann. Um dies zu ändern kann die Methode sleep durch die Methode yield ausgetauscht werden, die ich oben bereits vorgestellt habe.

6.3.1 Geschwindigkeitsvorteil

Das Programm wird durch die Änderung etwas schneller. Das ist dadurch zu erklären, dass die Prozesse schneller reagieren können, als wenn sie gezwungen sind zu warten. Das bringt vor allem den Vorteil, dass die Wartezeiten nicht durch eine Zeiteinheit bestimmt sind. achteile

6.4 Nachteil

Leider ist mir nach einigen Tests aufgefallen, dass das Programm einen großen Geschwindigkeitsunterschied zwischen einem und zwei Planeten, die berechnet werden, hat. Den Test habe ich auf einem dem Dualcore-Prozessor Athlon64 X2 5200+ von AMD ausgeführt.

Bei dem Test mit einem Planet, also einem Objekt das rechnet, war das Programm relativ schnell mit der Berechnung von 1000000 Schritten. Die Zeit betrug durchschnittlich 1.35 Sekunden. Während des Tests waren beide Prozessorkerne voll ausgelastet. Der

Thread, der für die Rechnung zuständig ist, hatte volle Last, da er entweder rechnete oder in einer Schleife den Prozessor an einen anderen Thread abgegeben hat. Der Thread, der die einzelnen Rechnungen synchronisiert, hatte auch volle Last, da er stets in einer Schleife mit dem abgeben des Prozessors an einen anderen Thread beschäftigt war. Bei den Tests habe ich den Thread für die grafische Darstellung ausgeschaltet um bessere Testergebnisse zu erhalten.

Den nächsten Test habe ich mit zwei Planeten ausgeführt, also mit zwei Rechentreads. Zur großen Enttäuschung war das Programm deutlich langsamer mit durchschnittlich 6.9 Sekunden für 1000000 Schritte. In dieser Programmausführung bekamen die beiden Rechentreads die CPU länger. Der Synchronisationsthread wurde fast immer verdrängt, da es Prozesse gab, die mehr auf der CPU zu tun hatten. Somit sollte die Geschwindigkeit für die gleiche Anzahl von Schritten ähnlich sein. Jedoch war diese signifikant langsamer, mehr als durch mehr benötigte Rechenzeit begründet wäre. Somit musste ein anderer Grund eine Rolle spielen, dass das Programm zu langsam lief.

Abgesehen von diesem schlecht ausgefallenen Test gab es einen weiteren Nachteil der aufgetreten ist. Es wird stets ein Prozessorkern mehr ausgelastet als es Rechnungen für Planeten gibt. Dabei sind Planeten, die fixiert sind, ausgenommen, da sie keinen eigenen Thread zur Berechnung benötigen. Dieser zusätzliche Kern wird durch den Thread zur Synchronisation ausgelastet, da dieser, wie schon beschrieben, in einer Schleife ist und ständig versucht seinen Platz auf dem Prozessor abzugeben.

6.4.1 Aufwand des Programms

Das Programm beinhaltet bei n Planeten n Rechnungen für jeden Planeten, angenommen es existieren keine fixierten Planeten. Für jeden Planeten muss eine Beziehung zu jedem anderen Planeten aufgebaut werden, um die Gravitation zwischen jedem Planetenpaar zu errechnen. Zusätzlich muss von jedem Planeten einzeln die neue Position berechnet und gespeichert werden. Also haben sie für die Gravitation einen Aufwand von $n \cdot (n - 1) \cdot x$ und für die Errechnung der Position und Speicherung $n \cdot y$, wobei x und y die benötigten Taktzyklen sind. Somit ist der Aufwand ungefähr $O(n + n^2)$.

Bei zwei Planeten und einem weiteren Kern der genutzt werden kann ergibt sich, dass der Aufwand $2 + 2^2 = 6$ ist. Da zwei Kerne zur Verfügung stehen, müsste der Aufwand durch zwei teilbar sein, also dann 3 betragen und um einen Faktor von 1.5 gesteigert sein im Vergleich zu einem Planeten. Dieser Faktor ist sehr viel höher und ist auch nicht durch Ungenauigkeiten in der Rechnung erklärbar.

6.4.2 Grund der schlechten Performance

Durch die benötigte Synchronisation sind die Threads, die rechnen, dazu gezwungen, die globalen Variablen in lokale Variablen zu kopieren. Diese lokalen Variablen werden von Java offensichtlich auch in den Speicher gelegt, wie im nächsten Kapitel gezeigt wird. Dieses Kopieren von Speicheradressen auf andere Speicheradressen kostet sehr viel Zeit. Wie weiter oben schon beschrieben, ist der Controller für den Speicher viel niedriger getaktet als der Prozessor. Somit brauchen diese Aktionen sehr lange. Dieses Kopieren

der globalen Variablen muss von jedem Rechenthread gemacht werden, um nicht die globalen Variablen zu überschreiben. Wäre das der Fall würde das Programm wieder nicht mehr deterministisch sein. Aus Gründen die ich oben genannt habe, kann jeweils nur ein Prozessorkern den Speicher benutzen. Als Konsequenz müssen die Threads gegenseitig aufeinander warten, bis der andere Thread fertig ist mit dem Kopieren der Daten. Nach dem Kopieren der Daten wird mit diesen teilweise noch gerechnet, was weitere Zugriffe auf den Speicher bedeutet. Nachdem alle Threads mit der Rechnung fertig sind, werden die Daten dann wieder auf die globalen Variablen geschrieben. Die Threads benutzen erneut den Speicher nur nacheinander. Um eine neue Rechnung anzufangen, setzen die Threads manche Variablen wieder auf einen Ausgangswert, was eine erneute Aktion auf dem Speicher bedeutet.

Aber auch die Synchronisation braucht Aktionen auf dem Speicher, um dort die Zustände der einzelnen Threads abzulegen. Bei jedem Zustandswechsel wird auf den Speicher geschrieben. Des Weiteren muss jeder Thread, wenn er wartet, den Zustand abfragen um entscheiden zu können, ob er weiterarbeiten darf. Das passiert jedesmal wenn der Thread Zeit auf der CPU bekommt, was nicht so sehr selten ist. Außerdem ruft der Thread, der zur Synchronisation dient, auch ständig nach den Zuständen der anderen Threads.

Das alles benötigt sehr viele Speicheraktionen. Man kann sogar sagen, dass das Programm mehr Zeit für Speicherzugriffe verbraucht als für das Rechnen. Da diese Speicherzugriffe die Dinge sind, die in den Threads am Meisten benutzt werden, ist das Programm unglaublich langsam.

In einem linearen Programm könnte man sich die ständigen Speicherzugriffe für die Synchronisation sparen. Außerdem würde auf dem Prozessor der Thread nicht dauernd wechseln, was auch einen Geschwindigkeitsvorteil bringt. So wäre ein lineares Programm schneller als dieses Programm.

Geschwindigkeitsvorteil durch Reduzierung von Speicheraktionen Im Anhang befinden sich zwei Klassen Calculation. Die eine Klasse ist übersichtlicher gestaltet als die andere, da Zwischenwerte in Variablen abgelegt werden. Die Andere ist darauf optimiert sehr wenig Variablen zu beschreiben, also den Speicher wenig zu belasten.

Testet man diese beiden verschiedenen Varianten mit zwei simulierten Planeten, so kommt folgendes Ergebnis heraus. Die nicht optimierte Klasse benötigt durchschnittlich 10.4 Sekunden für 1000000 Rechenoperationen. Die optimierte Klasse braucht ganze drei Sekunden weniger für das gleiche Ergebnis.

Daran kann man erkennen, dass Java wirklich die Variablen im Speicher anlegt. Des Weiteren sieht man, wie erheblich die Unterschiede sind, wenn ein paar Variablen mehr definiert werden in einem Abschnitt des Codes der viel benutzt wird. Die Zugriffszeit auf den Speicher ist einfach viel zu hoch.

7 Fazit

Es ist nicht sinnvoll eine solche Simulation mit mehreren Threads zu implementieren. Die Benutzung des Speichers wird unnötig hoch, zumal nicht parallel auf den Speicher zugegriffen werden kann. Wären die Abstände, in denen eine Synchronisation ausgeführt wird, sehr viel länger, würde das Speicherproblem reduziert werden, weil die Threads mehr Zeit ins Rechnen investieren anstatt in die Synchronisation oder Speicherzugriffe.

Ich denke Java ist keine geeignete Programmiersprache für solche Simulationen, da es gerade an rechenintensiven Stellen sinnvoll ist zu wissen, was der Prozessor bei den verschiedenen Befehlen macht. So könnte man an diesen Stellen häufige Speicherzugriffe verhindern, indem man manche Resultate in Registern speichert. Solche expliziten Angaben, welche Befehle der Prozessor ausführen soll, nennt man eingebettet in eine Hochsprache Inline-Assembler. Diese kann man zum Beispiel in C oder C++ benutzen. An stark verwendeten Stellen ist dies empfehlenswert, da man der CPU sehr gezielt sagen kann was gemacht werden muss. Dadurch kann an diesen Stellen eine sehr geringe Rechenzeit erreicht werden, was die Geschwindigkeit deutlich steigern kann. Diese Technik wird auch in Spielen benutzt, um diese signifikant zu beschleunigen.

Mir ist bei der Implementierung dieser Simulation in Java aufgefallen, dass ich gerne mehr wissen würde, was bei der Ausführung der verschiedenen Befehle von Java auf dem Prozessor passiert. Vor allem bei der Berechnung der Gravitation hätte ich bevorzugt mit Assembler gearbeitet, da es mir nicht als sinnvoll erscheint Daten die man nur kurz benötigt auf den langsamen Speicher zu schreiben. Ich denke man hätte die gesamte Rechnung samt Zwischenspeichern in Registern machen können, was das ganze Programm beschleunigt hätte.

Der Nachteil beim Schreiben von Assemblerprogrammen ist, dass diese nicht Plattformunabhängig sind. Daher passt dieses Konzept von Inline-Assembler auch nicht in die Programmiersprache Java. Diese ist gerade durch die Plattformunabhängigkeit ausgezeichnet.

Somit ist das Optimieren von Simulationen in allen Programmiersprachen sehr schwierig bis unmöglich. Unter Java wird dies jedoch noch erschwert durch das Konzept der Plattformunabhängigkeit.

8 Literaturverzeichnis

- Bernd Becker, R. Drechsler, P. Molitor: „Technische Informatik. Eine Einführung“, Pearson (2005)
- Rüdiger Brause: „Betriebssysteme. Grundlagen und Konzepte.“, 3. Auflage, Springer (2004)
- Paul Hermann: „Rechnerarchitektur“, 3. Auflage, Vieweg (2002)
- S. Jaksch: „Modellieren und Programmieren von nebenläufigen Prozessen : Beispiele in Java“, ISST (2003), Berlin
- Trutz Podschun: „Das Assembler-Buch, Grundlagen, Einführung und Hochsprachenoptimierung“, Addison-Wesley (2002), München
- Prof. Dr. Christian Siemers, Michael Eckert, Albert Lauchner: „Prozessor-Technologie“, tecchannel COMPACT 03/2004, PC-WELT (03/2004)
- Andrew S. Tanenbaum: „Moderne Betriebssysteme.“, 2. Auflage, Pearson (2003), München
- Andrew S. Tanenbaum: „Computerarchitektur. Strukturen - Konzepte - Grundlagen.“, 5. Auflage, Pearson (2006)
- Christian Ullenboom: „Java ist auch eine Insel. Programmieren mit der Java Standard Edition Version 5 / 6“, 6. Auflage, Galileo Computing (2007)

Erklärung Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

9 Anhang

9.1 Implementierung in Java

Die Bedienung ist unter `java Umlaufbahn -help` aufgeführt.

9.2 Optimierte Klasse Calculation