Informatik-Workshop – Wintersemester 2009/2010
Pascal Hahn, Marcus Körner, Alexander Mezler, Johannes Wachter

Zusammenfassung Multithreading

Schwierigkeiten von Multithreading

Deutlich höhere Komplexität

Nebenläufige Programme sind deutlich komplexer als sequenziell ablaufende Programme. Zum einen tauchen beim Multithreading eine Reihe neuer Probleme auf wie *Race Conditions*, *Deadlocks* und nicht deterministisches Verhalten, welche zu schwer reproduzierbaren *Bugs* führen können, zum anderen gehören parallele Standardalgorithmen zur Lösung eines Problems nicht unbedingt zum Erfahrungsschatz eines jeden Informatikers.

Menschliche Faktoren

Obwohl das menschliche Gehirn parallel arbeitet, denkt der Mensch insbesondere wenn er sich konzentriert oft nur sequenziell. Gefordert sind möglicherweise radikal neue Denkweisen und Methoden wie Programme entwickelt werden, da im Moment geeignete Abstraktionsmechanismen fehlen.

Standardisiertes Vorgehen

Consumer PCs haben einen regelrechten Multithreading-Hype ausgelöst, allerdings hat jede Programmiersprache eigene Konzepte, es gibt noch keinen Standard und diesen wird es möglicherweise auch nicht geben. Insbesondere in häufig genutzten Programmiersprachen wie Java sind neuartige Konzepte noch nicht implementiert.

Fehlendes Wissen (Studium)

Im Studium steht das sequenzielle Programmieren im Vordergrund. Multithreading ist komplex und erfordert ein gewisses Grundwissen und kann deshalb erst relativ spät gelehrt werden. Die im Studium verwendeten Sprachen sind darüber hinaus für Multithreading nicht optimal geeignet.

Wenig Tool-Support und Testen

Die große Mehrheit der verfügbaren Tools ist nicht für Multithreading ausgelegt, wie zum Beispiel *Profiler* oder *Debugger*. Auch auf dem Gebiet der *Unit-Tests* gibt es noch einiges zu tun, da *Context-Switches* in *Unit-Tests* reproduzierbare Ergebnisse nicht zulassen.

Komplizierte Laufzeit-Analyse und Wartung

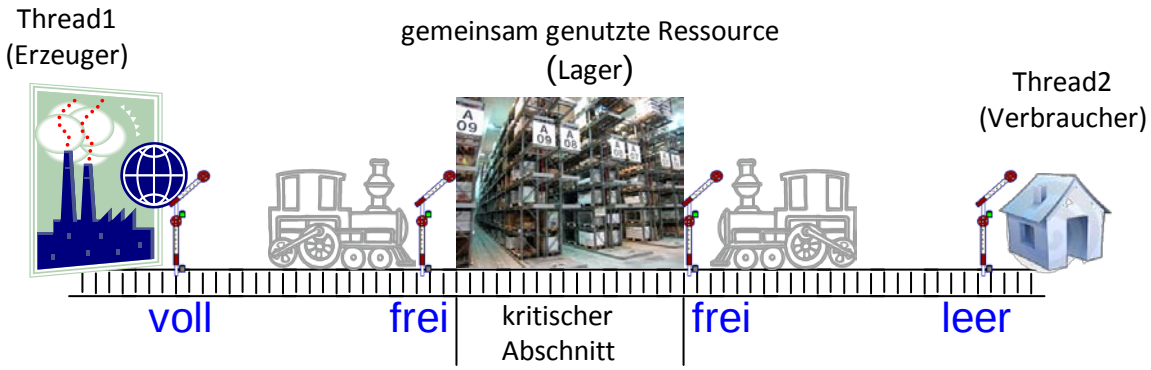
Eine Abschätzung der zu erwarteten Geschwindigkeitssteigerung durch Verwendung von Multithreading ist kompliziert. Es erfordert deshalb einer genauen Abschätzung, ob sich der Aufwand wirklich lohnt. Insbesondere da Multithreading den Wartungsaufwand eines Programms durch den komplizierteren Programmcode vergrößert.

Schlussfolgerung

Multithreading ist ein dringendes Problem an dem im Moment mit Hochdruck gearbeitet wird. Es ist allerdings nicht anzunehmen, dass in nächster Zeit ein Patent-Rezept erscheinen wird. Auch im Studium wird Multithreading in Zukunft eine immer wichtigere Rolle einnehmen müssen.

Bekannte Methodiken

Erzeuger-Verbraucher Problem



Locks/Semaphoren

Der Zugriff auf ein *Semaphor* ist eine unteilbare atomare Aktion.

Binäres Semaphor zum wechselseitigen Ausschluss (Absichern eines kritischen Abschnittes).

Zählendes Semaphor zur Sicherstellung gewisser Ereignisabläufe (*Synchronisation*).

Monitorkonzept

Zugriff auf kritischen Bereich wird gekennzeichnet, damit der *Compiler* die *Semaphoren* zum wechselseitigen Ausschluss setzen kann (in Java z.B. durch ***synchronized***).

Zum Sicherstellen gewisser Abläufe bzw. Synchronisation werden Zustandsvariablen mit den zugehörigen Operationen ***wait*** und ***signal*** verwendet. In Java stehen diese Zufallsvariablen bei der Verwendung von *Expliziten Locks* zur Verfügung. Der Standardmechanismus sieht hier lediglich ***wait*** und ***notify*** vor.

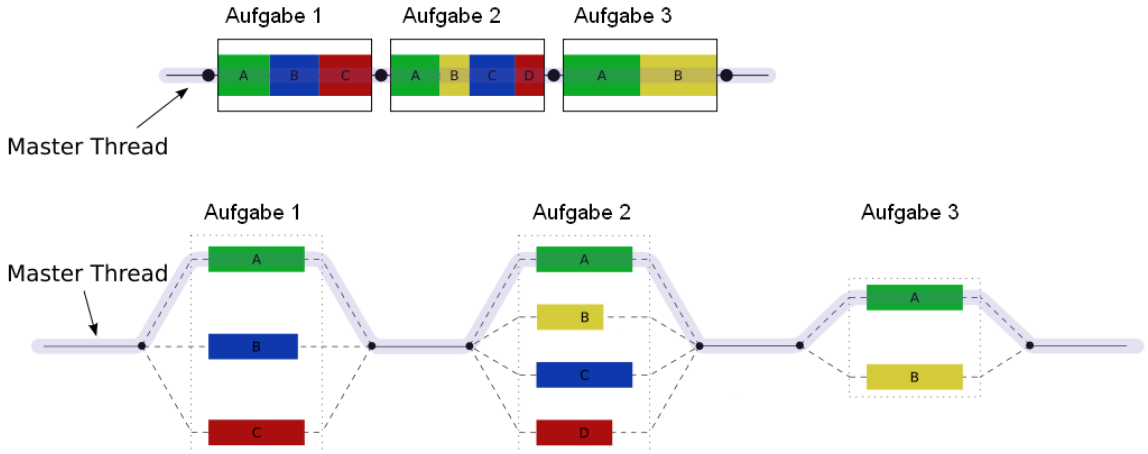
Message Passing

Versenden und Empfangen von Nachrichten. Geeignet für den Einsatz in verteilten Systemen. Asynchrone/synchrone Kommunikation mit gepuffertem/ungepuffertem Modus.

Massage Passing Interface (MPI) ist eine von *IEEE* standardisierte Kommunikationsbibliothek, welche Syntax, Semantik und Grundoperationen festlegt.

Fork/Join

Mit ***fork*** wird eine perfekte Kopie/Snapshot des Vaterprozesses erzeugt und mit ***join*** wird auf das Beenden dieses erzeugten Kindprozesses gewartet (Systemaufrufe unter Unix).



Divide-and-Conquer Ansatz:

Master-Thread für sequentiellen Teil eines Programms. Pro parallelisierbare Teilaufgabe wird ein Thread erzeugt. Am Ende des parallelisierbaren Blocks Zusammenführung / Synchronisation der einzelnen Threads.

Automatisierung von Multithreading

Automatische Parallelisierung

Compiler soll automatisch Parallelisieren. Es werden Schleifen parallelisiert. Dazu muss der *Compiler* eine Datenunabhängigkeitsanalyse durchführen, was nur für einfache Schleifen gut funktioniert. Um die Anzahl der parallelisierbaren Schleifen zu erhöhen, können einfache Code-Transformationen durchgeführt werden, um zu entscheiden, ob auch diese Schleifen datenunabhängig und somit parallelisierbar sind.

Nur eine geringe Anzahl von *Compilern* bietet diese Optimierung an und dann auch nur für recht einfache Schleifen.

Automatische Vektorisierung

Optimierter Code für *Vektorprozessoren* bzw. *Vektoreinheiten* in herkömmlichen Prozessoren. Datenunabhängigkeit der Iterationen muss gegeben sein. Durch geschickte Verteilung der Iterationen einer Schleife auf Vektorregister kann man mehrere Operationen (Iterationen) mit einer Rechenoperation der Vektorregister abarbeiten.

Parallele Sprachen

Idee:
Beim Entwurf der Sprache Parallelität berücksichtigen/einbauen.

Fortress:

Nachfolger für *FORTRAN* (angelehnt an mathematische Notationen), aber nicht abwärtskompatibel. Impliziter Parallelismus für bestimmte Sprachkonstrukte. Weiterhin die Möglichkeit explizit Threads anzulegen und kritische Bereiche abzusichern.

Unified Parallel C (UPC):

Auf C basierend, also eine Erweiterung von *ISO C*. Die Anzahl der parallelen Threads ist statisch (Festlegung zur Kompilierungszeit bzw. Programmstart möglich). *UPC* setzt das *Single Program Multiple Data* Modell um (d.h. jeder Thread führt das gleiche UPC-Programm aus). Lastverteilung auf mehrere Threads möglich. Methoden zur Absicherung von kritischen Bereichen bzw. zum Zugriff auf gemeinsame Daten vorhanden. Synchronisation durch Barrieren bzw. Split Phase Barrieren möglich.

OpenMP / JaMP

Bibliothek für *C/C++, Fortran* bzw. *Java* mit der man parallele Blöcke durch Direktiven kennzeichnet, die dann automatisch parallelisiert werden. Master-Thread für sequentiellen Programmteil. In parallelen Blöcken Lastverteilung auf mehrere Threads. Man kann die Sichtbarkeit und den Gültigkeitsbereichs von Variablen für die einzelnen Threads festlegen. Wechselseitiger Ausschluss und Synchronisation durch Barrieren sind möglich. *OpenMP* Unterstützung z.B. im *GCC* und *Visual Studio* eingebaut.

Fazit

Es gibt zwar Ansätze, welche dem Programmierer die Arbeit beim Automatisieren von Multithreading erleichtern/abnehmen, aber diese sind zur Zeit nur unzureichend. Einen guten Ansatz bietet hierbei OpenMP, aber auch hier muss man als Entwickler wissen, ob ein Block parallelisierbar ist. Die parallelen Sprachen fristen ein Nischendasein und werden nur in speziellen Bereichen eingesetzt. Multithreading wird wohl auch in Zukunft ein aktuelles Thema in der Software-Entwicklung bleiben und jeder Software-Entwickler wird sich damit auseinandersetzen müssen, um die steigende Anzahl der Prozessoren für seine Anwendung nutzen können.

Concurrent Java Programming

Prinzipien & Probleme

Synchronisation

Wenn mehrere Threads auf die gleichen Daten zugreifen, muss der Zugriff auf diese Daten synchronisiert ausgeführt werden, um sicherzustellen, dass Operationen konsistent für beide Threads angewendet werden.

In Java wird die Synchronisierung verschiedener Objekte mittels des ***synchronized*** Schlüsselwortes umgesetzt. Dieses bietet sowohl die Möglichkeit Methoden zu synchronisieren, als auch synchronisierte Blöcke zu erstellen, die ein beliebiges Objekt zur Synchronisation verwenden.

Zu beachten bei der Verwendung von ***synchronized***:

- Synchronisierte Methoden können die Skalierbarkeit negativ beeinflussen, wenn viele Felder voneinander unabhängig sind, da alle synchronisierten Methoden gleichzeitig blockiert werden.
- Synchronisierte Blöcke sollten so klein wie möglich gehalten werden, um eine gute Performance zu gewährleisten.
- Felder die unabhängig sind, sollten auch separat synchronisiert werden, zum Beispiel auf sich selbst.
- Felder die zusammengehören, müssen auf das gleiche Objekt synchronisiert werden.

Atomarität

Wenn mehrere Methodenaufrufe auf ein threadsicheres Objekt benötigt werden, um eine konsistente Änderung herbeizuführen, so müssen diese als eine einzige Operation betrachtet werden und auch als solche durchgeführt werden.

Zur Gewährleistung der Atomarität mehrerer Operationen wird ebenfalls Synchronisation benötigt.

Achtung: Auch wenn ein Objekt threadsicher ist, bedeutet dies nicht, dass mehrere Operationen darauf auch threadsicher beziehungsweise atomar sind.

Ein typisches Beispiel hierfür ist das *check-then-act Idiom*: Zuerst wird überprüft, ob eine Bedingung für ein Objekt erfüllt ist, danach wird das Objekt verändert. Diese beiden Operationen müssen atomar ausgeführt werden, da sonst ein nicht gewünschter Zustand oder ein Fehler auftreten kann.

Unveränderbarkeit

Sofern möglich, ist es in einer Parallelen Anwendung von Vorteil unveränderbare Objekte zu verwenden. Diese Objekte sind nach der Erzeugung threadsicher, da der Zustand nicht änderbar ist. Solche Objekte bieten den Vorteil, dass die Auszeichnung von Feldern mit ***volatile*** ausreicht diese und mögliche neue Instanzen sicher an andere Threads zu publizieren.

Explizite Locks

Explizite ***Locks*** sind eine andere, seit *JDK5* verfügbare Möglichkeit ***synchronized*** Blöcke und zusätzliche Lockobjekte zu vermeiden und einen flexibleren Mechanismus stattdessen zu verwenden.

Explizite ***Locks*** werden können als Instanzen der Klasse ***ReentrantLock*** verwendet werden. Diese Klasse bietet Möglichkeiten den Lock zu erhalten sowie ihn wieder freizugeben.

```
final ReentrantLock r = new ReentrantLock();

r.lock();
try {
      value = i;
} finally {
      r.unlock();
}
```

Wichtig ist dabei die Verwendung des *try-finally-Konstrukts* zur Sicherstellung der Lockfreigabe am Ende des Blockes. Bei ***synchronized*** geschieht diese Freigabe implizit.

Vorteile von Expliziten Locks:

Im Gegensatz zu ***synchronized*** ist das Warten auf einen ***Lock*** unterbrechbar. Somit wird vermieden, dass ein ***Lock,*** der nicht erhalten werden kann, die Unterbrechung des Threads verhindert.

Außerdem bieten Explizite Locks die Möglichkeit Deadlocks von vorneherein zu vermeiden ohne dabei auf dynamische Lock Sortierung setzen zu müssen.

Hierfür stehen die folgenden Methoden zur Verfügung:

```
r.tryLock();

r.tryLock(TIMEOUT, TimeUnit.SECONDS);
```

Beide Methoden versuchen sofort den Lock zu bekommen. Wenn sie erfolgreich sind kehren beide mit **true** zurück.

Kann der Lock nicht erhalten werden, kehrt ***tryLock()*** sofort mit **false** zurück. ***tryLock()*** mit Timeout versucht, solange der Timeout nicht abgelaufen ist, den Lock zu erhalten und kehrt erst danach mit **false** zurück, wenn er immer noch nicht erhalten werden kann.

Die Performance von ***ReentrantLock*** im Vergleich zu ***synchronized*** ist im *JDK5* um einiges größer, während in *JDK6* ***synchronzied*** von der reinen Geschwindigkeit aufgeholt hat und der Abstand geringer wurde.

Eine Erweiterung des normalen expliziten ***Locks*** wird durch die Klasse ***ReadWriteLock*** definiert. Anstatt lediglich einen Lock zu verwenden bietet diese die Möglichkeit einen exklusiven ***WriteLock*** sowie einen ***ReadLock*** den mehrere Threads erhalten können zu verwenden. Hierdurch kann die Performance vor allem dann erhöht werden, wenn die Schreibzugriffe weniger häufig als die Lesezugriffe stattfinden.

Beispiel der Verwendung in einer ***put***- und ***get***-Methode:

```
public ValueStore() {
	rrw = new ReentrantReadWriteLock();
}

// weitere Methoden

public void put(final int i) {
	final WriteLock writeLock = rrw.writeLock();
	try {
		value = i;
	} finally {
		writeLock.unlock();
	}
}

public int get() {
	final ReadLock readLock = rrw.readLock();
	try {
		return value;
	} finally {
		readLock.unlock();
	}
}
```

Task Execution

Das Prinzip der *Task Execution* basiert darauf, dass in Anwendungen häufig verschiedene Aufgaben auf eine Menge von Eingabedaten angewendet werden sollen. Dies wird im Normalfall durch eine Schleife gelöst. Da die einzelnen Eingabedaten aber oft unabhängig sind und die darauf angewandten Operationen ebenfalls, besteht die Möglichkeit diese Ausführungen parallel zu implementieren. Dabei wird für jedes Eingabedatum ein Task erstellt, der eine oder mehrere Aufgaben kapselt, die auf dieses Datum angewandt werden sollen. Mehrere dieser Tasks können dann parallel ausgeführt werden. Die Ergebnisse werden dann am Ende aggregiert.

Java stellt seit JDK5 das ***Executor-Framework*** als Umsetzung des *Task-Execution-Prinzips* zur Verfügung. Beim ***Executor-Framework*** handelt es sich um Klassen und Konzepte die es erlauben ohne das direkte Arbeiten mit Threads die Vorteile zu nutzen und parallele Aufgaben ausführen zu lassen.

Das Hauptinterface des Frameworks ist ***Executor***. Das ***Executor*** Interface besitzt lediglich eine Methode, um eine Aufgabe auszuführen. Im Normalfall werden ***Executor***-Implementierungen durch Thread Pools abgebildet, die sich durch verschiedene Eigenschaften auszeichnen:

- **Cached Thread Pool**: Soll ein Task ausgeführt werden und es ist kein unbeschäftigter Thread vorhanden, wird ein neuer Thread erzeugt, der diesen Task ausführt. Nach der Ausführung eines Tasks geht ein Thread in den *Idle-Zustand* über und ist bereit weitere Tasks anzunehmen.
- **Fixed Thread Pool:** Es steht eine festgelegte Menge an Threads zur Verfügung, auf denen Tasks ausgeführt werden können. Sind alle Threads beschäftigt, wird ein neuer Task in eine Queue eingereiht bis ein Thread bereit ist ihn auszuführen.
- **Single Thread Pool:** Ein Pool der lediglich aus einem Thread besteht, ein Spezialfall des Fixed Thread Pool.
- **Scheduled Thread Pool:** Verhält sich wie ein Fixed Thread Pool besitzt aber die Möglichkeit Tasks mit Verzögerung oder periodisch auszuführen.

Der Vorteil der Verwendung eines ***Executor*** ist, dass dieser konfiguriert werden kann und damit eine genaue Kontrolle über die Anzahl der verwendeten Threads besteht, ohne die Verwaltung dafür übernehmen zu müssen. Stirbt ein Thread in einem ***Executor*** unbeabsichtigt, so wird dieser automatisch neu erzeugt, damit die gewünschte Anzahl an Threads wieder zur Verfügung steht.

Um die Verwendung von ***Executor*** in der Praxis zu vereinfachen, steht außerdem das Interface ***ExecutorService*** zur Verfügung. Diese Klasse erweitert ***Executor*** und bietet Möglichkeiten, die in der Praxis benötigt werden, um das ***Executor-Framework*** sinnvoll einzusetzen.

Hierzu zählt die Möglichkeit für Tasks mit Rückgabewerten, sowie die Bereitstellung eines *Lifecycles* für den ***ExecutorService*** selbst.

Im Folgenden zwei Beispiele für die Erstellung von Tasks. Task1 wird als ***Runnable*** ohne Rückgabewert implementiert. Task2 wird als ***Callable***, welches einen String zurückliefert implementiert.

```
final Runnable task1 = new Runnable() {

	@Override
	public void run() {
		// do something
	}
};
```

```
final Callable<String> task2 = new Callable<String>() {
	@Override
	public String call() throws Exception {
		final String result = ""; // do something
		return result;
	}
};
```

Im Falle der Verwendung von ***Callable*** als Basisinterface erhalten wir sofort ein ***Future***-Objekt, welches das Ergebnis des Tasks kapselt. Die ***get***-Methode, die das Ergebnis zurückliefert, blockiert dabei solange bis ein Ergebnis vorliegt oder der Task abgebrochen wurde.

```
final Future<String> result = executor.submit(task2);

result.get(); // Blockt bis der Task ein Ergebnis liefert oder abbricht
```

Der *Lifecycle* des ***ExecutorService*** gliedert sich in drei Phasen:

Running ist der Startzustand jedes ***ExecutorService,*** der sofort nach der Initialisierung angenommen wird. Neue ***ExecutorService*** Instanzen werden normalerweise mittels der Utility-Klasse ***Executors*** oder einer der ***newXXX(…)*** Methoden erzeugt.

Shutting Down wird erreicht, wenn das Programm den Service über die ***shutdown***- oder die ***shutdownNow***-Methode zum herunterfahren auffordert. ***Shutdown*** lässt hierbei noch vorhandene Tasks fertig ausführen (auch in der *Queue* befindliche), nimmt aber keine neuen mehr an. ***ShutdownNow*** versucht den Service sofort durch Abbrechen der laufenden Tasks zu beenden und liefert die in der Queue befindlichen Rest-Tasks als Liste zurück.

Versucht man einem ***ExecutorService,*** der im *Shutting Down Zustand* ist, einen Task zu übergeben, erhält man eine ***Exception***.

Sind alle laufenden Tasks beendet, geht der Service in den **Terminated** Zustand über.

Es ist sinnvoll vor dem Beenden eines Programmes nicht nur den ***shutdown*** auszulösen sondern auch auf die Terminierung zu warten. Dies ist zum Beispiel durch ***awaitTermination*** mit einer Timeout möglich. Ob der Service terminiert ist, lässt sich dann mittels ***isTerminated*** erfragen.

Atomare Datentypen

Java bietet seit JDK5 mittels der Klassen ***AtomicInteger, AtomicLong, AtomicBoolean*** die Möglichkeit einfache Counter und Ähnliches mittels dieser Typen threadsicher mit atomaren Methoden zum inkrementieren, … umzusetzen.

Parallele Collections

In JDK5 kamen ebenfalls einige Collection-Klassen hinzu, die speziell für parallele Anwendung optimiert sind.

ConcurrentHashMap: Bietet eine deutlich höhere Performance als eine synchronisierte ***Map***. Außerdem wird die atomare ***putIfAbsent-Methode*** zur Verfügung gestellt. Die beiden wichtigsten sind im Folgenden kurz beschrieben.

BlockingQueue: Perfekt für Producer-Consumer-Problem. Die ***put***-Methode blockiert, sobald die Queue gefüllt ist; die ***get***-Methode blockiert, sobald die Queue leer ist.

Best Practices

1. Sichtbarkeit von Feldern sollte soweit wie möglich eingeschränkt werden.
    a. Was nicht öffentlich sein **MUSS,** sollte als ***private*** deklariert werden.
    b. Kapselung verhindert unerwünschte Zugriffe von anderen Threads.
    c. Kapselung ermöglicht die Verwendung nicht threadsicherer Klassen.
2. Veränderbarkeit minimieren
    a. Was nicht änderbar sein **MUSS,** sollte als ***final*** deklariert werden.
    b. Unveränderbare Objekte sind automatisch threadsicher.
3. Synchronisation
    a. Jeder synchronisierte Block sollte so klein wie möglich gehalten werden.
    b. Felder, die unabhängig sind, müssen von verschiedenen ***Locks*** geschützt werden, um schlechte Performance zu verhindern.
    c. Felder, die abhängig sind, müssen von dem gleichen ***Lock*** geschützt werden.
    d. Synchronisation sollte in der Klasse gekapselt sein und nicht dem Benutzer der Klasse überlassen werden.
4. Dokumentation
    a. Threadsicherheit und Nicht-Threadsicherheit müssen dokumentiert werden.
    b. Synchronisierungsregeln für Klassen müssen dokumentiert werden.
5. ***Executor-Framework***
    a. Das ***Executor-Framework*** sollte der Verwendung von Threads wenn möglich vorgezogen werden.
    b. Im Speziellen sollte der ***ScheduledThreadPoolExecutor*** der Verwendung von ***TimerTask*** vorgezogen werden.

Tipps

Bei der Entwicklung von parallelen Programmen sollte die JVM immer mit dem Parameter ***–server*** gestartet werden. Durch diese Option führt die JVM aggressivere Optimierungen am Code und

Bytecode, durch welche dabei helfen Synchronisierungs- und Sichtbarkeitsprobleme früher zu entdecken.

Statische Codeanalyse Werkzeuge wie *FindBugs* bieten die Möglichkeit viele Programmierfehler zu finden. Unter anderem auch nicht freigegebene oder unsicher freigegebene ***Locks***.

Multithreading mit C#

In C# gibt es neben dem klassischen Programmiermodell mit Threads, ab .NET 4.0 die Parallel Extensions. Es handelt sich dabei um ein Framework, welches die Entwicklung von nebenläufigen Programmen wesentlich vereinfachen soll.

Threads in C#

Threads in C# sind vergleichbar mit Threads in Java, weshalb sich beide Programmiermodelle sehr ähnlich sind. Man erzeugt eine neue Instanz der Klasse Thread und übergibt im Konstruktor die Methode, die später vom Thread ausgeführt werden soll.

```
Thread thread = new Thread(delegate() {
    // Funktionsrumpf
    Console.WriteLine("Hello world!");
});
```

Man startet den Thread mit der Methode `Start` und wartet auf das Beenden eines Threads mit `Join`. Die Synchronisation von Variablen wird über das Schlüsselwort `lock` vorgenommen oder über threadsichere Methoden wie `Interlocked.Add` oder `Interlocked.Increment`.

```
long SumPrimesTraditional(int number) {

    // Ermitteln der Threadanzahl
    int threadCount = 4;
    int numbersPerThread = number / threadCount;
    Thread[] threads = new Thread[threadCount];

    long sum = 0;

    for (int i = 0; i < threadCount; i++) {
        threads[i] = new Thread(delegate() {
            for (int num = i * numbersPerThread; num < (i + 1) * numbersPerThread; num++) {
                if (isPrime(num))
                    // Threadsicheres Addieren
                    Interlocked.Add(ref sum, i);
                }
            });
        threads[i].Start();
    }

    // Warten auf Ende aller Threads
    for (int i = 0; i < threadCount; i++) {
        threads[i].Join();
    }
    return sum;
}
```

Problematisch an diesem Ansatz ist, dass die Anzahl Threads vom Programmierer selbst vorgegeben werden muss. Die optimale Anzahl Threads ist allerdings von der verwendeten CPU abhängig. Terminiert ein Thread früher als ein anderer, können vorhandene Kerne ungenutzt bleiben: es gibt keine dynamische Arbeitsteilung. Darüber hinaus verursachen Threads durch Context-Switches relativ viel Overhead.

Parallel Extensions

Mit .NET 4.0 wurde das Paralle Extensions Framework in C# eingeführt. Es besteht aus 3 Hauptkomponenten:

- Task Parallel Library
- Coordination Data Structures
- Parallel LINQ

Task Parallel Library

Bei der Task Parallel Library (TPL) handelt es sich um eine Bibliothek die das Konzept der Threads durch leichtgewichtige Threads, sogenannte Tasks ersetzt. Tasks sind Arbeitseinheiten die unabhängig voneinander ausgeführt werden können und von einem internen Taskmanager automatisch auf alle verfügbaren Kerne verteilt werden. Tasks können an den gleichen Stellen wie Threads eingesetzt werden, auch das Programmiermodell unterscheidet sich nicht wesentlich. Tasks werden durch eine Instanz der Klasse Task<Rückgabewert> erzeugt, wobei im Konstruktor die auszuführende Methode übergeben wird.

```
Task<int> task = new Task<int>(delegate() {
    // Funktionsrumpf
    return doWork();
});
```

Die Anzahl der Tasks kann dabei wesentlich höher gewählt werden, als die Anzahl der Threads. Tasks werden mit der Methode `Start` gestartet und der Rückgabewert eines Tasks kann über `Task.Result` abgefragt werden. Die Methode `WaitAll(Task[] tasks)` wartet auf das Beenden aller übergebenen Tasks.

```
long SumPrimesTasks(int number) {

    // Anzahl der Tasks festlegen
    int threadCount = 100;
    int numbersPerThread = number / threadCount;
    Task<long>[] tasks = new Task<long>[threadCount];

    // Tasks erzeugen
    for (int i = 0; i < threadCount; i++) {
        tasks[i] = new Task<long>(delegate() {
            long sum = 0;
            for (int num = i * numbersPerThread; num < (i + 1) * numbersPerThread; num++) {
                if (isPrime(num))
                    sum += num;
            }
            return sum;
        });
        tasks[i].Start();
    }

    // warten bis alle Tasks terminiert sind
    Task.WaitAll(tasks);

    // Summe berechnen
    long totalSum = 0;
    for (int i = 0; i < threadCount; i++) {
        totalSum += tasks[i].Result;
    }
    return totalSum;
}
```

Zu der TPL gehört ebenfalls eine parallel For-Schleife die auf dem Task-Konzept aufbaut und damit ebenfalls eine dynamische Arbeitsteilung aufweist.

```
private static long SumPrimesBelow(int number) {
    long sum = 0;
    Parallel.For(0, number, delegate(int i) {
        if (isPrime(i)) {
            Interlocked.Add(ref sum, i);
        }
    });
    return sum;
}
```

Coordination Data Structures

Bei den Coordination Data Structures (CDS) handelt es sich um eine Sammlung von threadsicheren Datenstrukturen und Synchronisationsprimitiven.

Zur Implementierung des Producer/Consumer Patterns enthält die CDS beispielsweise die `BlockingCollection` Klasse, auf die mit den Methoden `Add` und `Take` zugegriffen werden kann. Zusätzlich sind threadsichere Stacks, Warteschlangen und Wörterbücher enthalten.

Bei den Synchronisationsprimitiven ermöglichen die Klassen `SpinWait` und `SpinLock` ein aktives Warten, um aufwendige Context-Switches zu vermeiden.

Um eine Variable außerhalb eines Konstruktors genau einmal zu setzen, gibt es die Klasse `WriteOnce` und `LazyInit` ermöglicht das threadsichere Initialisieren eines Objektes zum Zeitpunkt des ersten Zugriffs.

Für Fork/Join Szenarien ist die Klasse `CountDownEvent` geeignet. Diese ermöglicht es den Hauptthread solange zu blockieren, bis eine bestimmte Anzahl von Worker Threads terminiert ist.

Parallel LINQ

Mit LINQ bietet C# die Möglichkeit auf Objekten SQL-ähnliche Abfragen durchzuführen. Mit dem Parallel Extensions Framework wurde diese Möglichkeit dahin gehend erweitert, dass LINQ Abfragen automatisiert, durch das Aufrufen der Methode `AsParallel`, parallelisiert werden können.

```
long SumPrimesBelowPLINQ(int to) {

    // Primzahlen auflisten
    var primes = from number in Enumerable.Range(0, to).AsParallel()
                 where isPrime(number)
                 select (long) number;

     // Summe bilden
    return primes.Sum();
}
```

Vorsicht: bei PLINQ kann sich die Reihenfolge der Elemente in der Liste beliebig ändern. Ist die Reihenfolge wichtig, muss man die Liste mit `OrderBy` sortieren.

Best Practices

Falls möglich sollte mit Erscheinen der .NET 4.0 Plattform auf Threads verzichtet werden und stattdessen auf Tasks zurückgegriffen werden.

An den Stellen wo es sich anbietet, sollte aus Gründen der Verständlichkeit des Codes auf die parallele For-Schleife zurückgegriffen werden. Anwendungsszenarien sind beispielsweise Matrix-Multiplikationen.

Datenstrukturen und Primitiven der Coordination Data Structures sollten eingesetzt werden, falls die vorhandene Funktionalität nicht ausreicht oder man Low-Level Optimierungen durchführen will.

Ist es möglich einen Algorithmus durch eine LINQ Abfrage zu formulieren, kann man mit sehr geringem Aufwand die Abfrage automatisiert parallelisieren lassen.

Actor Model in Scala

Das mathematische Actor-Model wurde 1973 von Carl Hewitt, Peter Bishop und Richard Steiger im Paper "A Universal Modular ACTOR Formalism for Artificial Intelligence" dargelegt und beschreibt einen Actor als grundlegende Einheit nebenläufiger Programmierung. Ursprünglich war es für das Gebiet der Künstlichen Intelligenz ausgelegt und beschreibt die parallel ablaufende Kommunikation von Aktoren (oder auch Virtuelle Prozessoren) auf Basis von Message Passing.

Das Actor-Model wurde vor allem durch die Implementierung der Actors in Erlang bekannt. Scala eine neu aufkommende Sprache für die JVM, die objektorientierte und funktionale Sprachelemente vereint, hat eine sehr Erlang ähnliche Implementierung dieses Models übernommen.

Ein Actor kann ähnlich der Objektorientierung wie ein eigenständiges Objekt verstanden werden, mit der Ausnahme das es explizit parallel und damit autonom läuft.

Das Prinzip dahinter besagt vor allem das die Reihenfolge der eintreffenden Nachrichten keine Auswirkung haben darf.

Beim Eintreffen einer Nachricht kann ein Actor...

- ...eine endliche Anzahl von Nachrichten an andere Actors schicken
- ...eine endliche Anzahl neuer Actors erzeugen
- …das Verhalten beim Erhalt der nächsten Nachricht bestimmen

Grundkonzepte des Actor-Models

Alles ist ein Aktor! Das Actor-Model besagt, dass ein Actor seinen eigenen Zustand und sein Verhalten kapselt. Hiermit ähnelt das vorgeschlagene Model viel mehr dem ursprünglichen OOP Konzept als Klassen das heutzutage tun.

Die Interaktion zwischen mehreren Actors findet rein auf Basis des Messag-Passing Konzepts statt. Bei Scala wird vor allem empfohlen keinerlei mutable Datentypen in einer Nachricht zu verschicken. Hierdurch umgeht man das Prinzip eines Shared Memory komplett und hat den Vorteil das keinerlei Synchronisation oder Koordination stattfinden muss. Dies fördert zudem die Vermeidung von Race Conditions.

Grundregel:

Give each Actor just a couple of responsibilities and use messages, (usually in the form of a case class or case object) to delegate those responsibilities to other Actors!

Scala

Scala ist eine moderne Sprache die für die JVM konzipiert ist. Sie vereint Konzepte von funktionalen sowie objektorientierten Sprachen. Kernelemente sind vor allem:

- Pattern-Matching
- Sehr guter Support für immutable Datenstrukturen (Listen, Maps)
- Statische Typisierung
- Native XML Support (auf Basis von XML Literalen)

- Umsetzung des Actor-Models für parallele Programmierung

Scala bietet zwei verschiedene Möglichkeiten für die Umsetzung eines Actors. Er kann entweder rein Event-basiert sein und wird dann über einen Thread-Pool ähnlich dem Executor-Framework gehandhabt, oder er kann einem nativen Thread entsprechen.

Einstieg in Scala Actors

Ein simpler Actor kann auf zwei Arten implementiert werden. Entweder durch direktes vererben des Traits Actor (oder als Mixin mittels **with** Keyword) oder durch Nutzung des Konstrukts **actor {}** welches direkt einen anonymen Aktor einer Variable oder einem Value zuweist.

Das Versenden einer Nachricht an einen Actor erfolgt über den Bang-Operator **!** wie man den folgenden zwei Beispielen entnehmen kann.

```
package actortest
import scala.actors.Actor
class HelloWorldActor extends Actor {
        def act {
                receive {
                        case 'hello => println ("Hello World")
                }
        }
}
object Main {
        def main (args:Array[String]) {
                val helloWorldActor = new HelloWorldActor
                helloWorldActor.start
                helloWorldActor ! 'hello
        }
}
```

```
package actortest
import scala.actors.Actor._

object Main {
    def main (args:Array[String]) {
        val helloWorldActor = actor {
            receive {
                case 'hello => println ("Hello World")
            }
        }
        helloWorldActor ! 'hello
    }
}
```

Ein anonymer Actor muss nicht erst mit Aufruf der Methode .start gestartet werden! Anonyme Actors können gut dazu verwendet werden, um kürzere Codeanteile von Algorithmen zu parallelisieren.

Case Classes & Pattern Matching

Case Classes und Pattern Matching sind zwei gängige Konzepte, die für vieles Anwendung finden in der Programmiersprache Scala. So auch beim Actor-Model. Hier bilden diese zwei Konzepte die Grundsteine für die Kommunikation zwischen Aktoren. Case Classes werden hier genutzt, um Nachrichtentypen festzulegen. Dies ermöglicht es beim Pattern Matching der **react** und **receive** Methoden sehr einfach auf die eintreffende Nachricht zu prüfen.

```
case class Message (attribute1, attribute2)
case class MessageResponse (result)
case class Exit

react {
  case Message (a1, a2) => println (a1 +“ “ + a2)
  case Exit => System.exit (0)
  case _ => println ("Unknown Message")
}
```

Request Response Pattern

Ein gängiges Verhalten bei der Parallelisierung mittels Message Passing ist, dass einer Anfrage eine Antwort an den ursprünglichen Sender folgt. Hierzu hat man bei den Actors in Scala mittels eines Tupels den Actor mitgeschickt, an den der Response erfolgen sollte.

```
Actor ! (self, Message)
...
case (sender, Message) => sender ! Response(...)
```

Aufgrund der Häufigkeit dieses Musters wurde diese Funktionalität nun direkt im Actor zur Verfügung gestellt. Wird also nun eine Nachricht empfangen, kann mittels dem Objekt **sender** oder der Methode **reply** eine direkte Nachricht an den Actor geschickt werden von dem diese Anfrage stammt.

```
case Message => sender ! Response
oder
case Message => reply (Response)
```

Futures

Futures sind ein Konzept zur "Lazy Evaluation" von parallel berechneten Ergebniswerten, also einer asynchronen Berechnung.

Erwartet man eine Antwort auf eine Nachricht an einen Aktor, kann diese sehr einfach über die Methode !! anstatt ! in eine Future Variable umgelenkt werden. Der Vorteil hierbei ist natürlich, dass der aufrufende Code nicht selbst ein Actor sein muss. Die Funktion schickt die besagte Nachricht an den Actor und liefert unmittelbar sozusagen als Platzhalter für das Ergebnis ein Future zurück.

```
val awaitResult = actor !! ProcessData (data)
...
val result = awaitResult ()
```

Um den Wert auszulesen muss die Methode apply auf das Future aufgerufen werden. Die Kurzform in Scala hierfür ist einfach nur **future ()**. Allerdings ist hierbei zu beachten, dass die Operation blockt, solange bis ein Ergebnis geliefert wird.

Man hat ebenfalls die Möglichkeit zu prüfen, ob ein Future schon gesetzt wurde, bevor man in die blockende Warteoperation springt. Dies ist mittels der Methode .isSet () möglich.

Futures eignen sich sehr gut zum Joinen von parallelen Ergebnissen über die Funktion map, die von Sequenzen in Scala angeboten werden. Siehe hierzu das nachfolgende ParallelMapper Beispiel.

ParallelMapper Beispiel

```
case class SearchTrip (val trip : (Place, Place, Date))
case class SearchTripResponse (val flights:Seq[specification.Itinerary])

object ParallelMapper extends Mapper {

def map (trips: Seq[(Place, Place, Date)]) = trips.map ( singleTrip => {
    actor {
      react {
        case SearchTrip(trip) =>
        reply    (SearchTripResponse(searchOneway    (trip._1,    trip._2,
trip._3)));
            }
          } !! SearchTrip (singleTrip)
     }).map (future => {
            val ft = future ().asInstanceOf[SearchTripResponse]
            ft.flights
            }
        )
     }
   )
```

Unterschied zwischen react und receive

Der Unterschied zwischen **react** und **receive** ist, dass bei der Verwendung von receive innerhalb eines Actors die Umsetzung auf native JVM Threads erfolgt. Dies hat zufolge dass das Umschalten eines Threads tatsächlich auf Hardwareebene erfolgt und ist somit um einiges kostspieliger bezüglich der Rechenzeit.

Im Normalfall sollte man daher immer auf **react** zurückgreifen beim Implementieren eines Actors. Dieser nutzt einen dynamischen Shared-Pool an Hardware-Threads aus und muss beim Umschalten diese nur auf Softwareebene austauschen. Die Nutzung von **react** ähnelt daher einem leichtgewichtigen Event- oder Nachrichtensystem, das automatisch verteilt wird über eine dem System mögliche Anzahl an Threads. Hierdurch erhält man auch eine bessere Ausnutzung der Ressourcen.

RemoteActor

Mit Hilfe von RemoteActors erhält man eine simple Lösung, um die Kommunikation von verteilten Actors über TCP zu ermöglichen. Hierzu sind nur wenige Methoden nötig, um einen Actor in einen RemoteActor umzuwandeln.

Das folgende Beispiel zeigt wie simpel man einen normalen Actor in einen Server-Actor umwandeln kann:

```
val server = actor {
    // listen on port 5050
    alive (5050)
    // register actor self on identifier symbol
    register ('ServerIdentifier, self)
    react { ...
```

Der passende Client hierzu würde sich wie folgt auf diesen Server-Actor verbinden:

```
val client = actor {
    // select liefert ProxyObjekt auf den RemoteActor
    val server = select (Node("localhost", 5050), 'ServerIdentifier)
    react { ...
```

Nach dem Ausführen des **select** erhält man ein ProxyObjekt auf den RemoteActor und kann nun mittels **server ! Message** Nachrichten an den Server schicken. Dieses sehr einfache Konzept bringt allerdings auch mit sich, dass es bisher kein Sicherheitskonzept für diese Kommunikation gibt.

Kommende Neuerungen in Scala 2.8

Mit der kommenden Version 2.8 von Scala gibt es auch einige Neuerungen bei der Implementierung des Actor-Models. Vor Allem hat man nun die Möglichkeit anstelle eines Actors den neuen Reactor Trait zu nutzen. Dieser ermöglicht einen noch geringeren Verbrauch an Ressourcen (leichtgewichtiger Actor). Ermöglicht wird dies durch einige Einschränkungen gegenüber den herkömmlichen Actors:

- Kein impliziter **sender** mehr verfügbar (Hierzu muss nun der ReplyActor genutzt werden)
- Nur noch Event basiert, sprich Nachrichtenempfang über **react** (kein receive mehr)
- Weniger Zustände werden gespeichert (Zustände die nötig waren für native Threads)
- Keine **!!** und **!?** Methoden mehr zum synchronen Senden oder Senden mit Future als Rückgabe

Einer der Vorteile dieser Neuerung ist, dass diese Eigenschaften von vielen Systemen nicht benötigt wurden und man nun durch die Speichereinsparung eine Vielzahl mehr an Reactors erzeugen kann als an Actors.

Fazit

Actors in Scala bieten eine sehr gute Grundlage für Skalierbarkeit, da sie mittels Event basierter Umsetzung die Möglichkeit bieten automatisch so viele Cores und Threads auszunutzen wie ihnen zur Verfügung stehen. Durch das Umgehen von Shared Memory spart man sich viel Mühe aufwändige Locking-Mechanismen zu implementieren oder in langen Debug-Sessions Bugs in

nebenläufigem Code zu suchen. Durch das Message-Passing Prinzip erhält man zusätzlich auch noch eine Ideale Grundlage, um ein System in parallel laufende Module zu trennen und muss diese nicht erst mittels eins Service-Busses verknüpfen.

Selbstverständlich gibt es auch Nachteile wie den fehlenden Support in einigen etablierten Programmiersprachen. Zudem bricht es stark mit den vorhandenen Konzepten und stellt so vorhandene Programmiermodelle paralleler Programme auf den Kopf.

Mit dem ständigen Anstieg von Cores, Threads und Prozessoranzahl sollte man dennoch Actors als ein gutes Konzept zur Grundlage für skalierbare Systeme im Kopf behalten.

Der Quelltext eines Parallel-Merge-Sorts mit Aktoren kann unter

http://gist.github.com/306844

aufgerufen werden.

Quelltexte der Übungsaufgaben

http://github.com/jwachter/iws-fileindex-example

Weiterführende Literatur

BENGEL, Günther et al. – **Masterkurs Parallele und Verteilte Systeme** (ISBN 978-3-8348-0394-8)

BLOCH, Josh – **Effective Java: A Programming Language Guide** (ISBN 978-0-321-35668-0)

GOETZ, Brian – **Java Concurrency in Practice** (ISBN 978-0-321-34960-6)

ODERSKY, Martin et al. – **Programming in Scala** (ISBN 978-0-9815316-0-1)

SUBRAMANIAM, Venkat – **Programming Scala** (ISBN 978-1-93435-631-9)

DENNIS, Alan - **.NET Multithreading** (2002) (ISBN 978-1930110540)