Universität Passau
Fakultät für Mathematik und Informatik

Multicore-Parallelität in Haskell

Ausarbeitung im Rahmen des Seminars *Multicore-Programmierung*

Stefan Boxleitner

20. Januar 2011

Betreut von Dr. Armin Größlinger

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren gab es einen Umbruch in der Prozessortechnologie. Aufgrund von Einschränkungen der Halbleitertechnik geht der Trend weg von Prozessoren, deren Taktrate grenzenlos erhöht wird. Stattdessen werden mehr und mehr Prozessorkerne bei annähernd gleich bleibender Taktrate auf einem Sockel platziert, die sich einen gemeinsamen Hauptspeicher teilen. Um die Mehrkern-Architektur optimal auszunutzen gibt es eine Reihe von Konzepten. Zunächst sollte unterschieden werden zwischen konkurrierender und paralleler Programmierung.

Konkurrierende Programme führen gleichzeitig mehrere möglicherweise voneinander unabhängige Aufgaben aus. Ein Beispiel wäre ein Datenbanksystem, das für Datenbankzugriffe und den Abgleich mit anderen Datenbanken jeweils eigene Komponenten besitzt. Konkurennte Programme müssen nicht zwangsläufig auf mehreren Prozessorkernen laufen, können aber möglicherweise dadurch in ihrer Ausführung beschleunigt werden.

Parallele Programme hingegen lösen in der Regel ein einzelnes Problem. Beispielsweise die Berechnung eines Bildes in der Videobearbeitung. Wenn man diese Berechnung pro Pixel durchführt und auf die vorhandenen Prozessorkerne verteilt, so wird die Laufzeit der gesamten Berechnung möglicherweise verkürzt. Auch Parallele Programme müssen nicht zwangsläufig auf einem Mehrkern-Prozessor ausgeführt werden [OSG11].

Eine weitere Klassifizierung kann durch den Grad an Einflussnahme des Programmierers auf die Verteilung von Aufgaben und Daten auf Berechnungseinheiten, d. h. hier Prozessorkerne.

Explizite Parallelität delegiert die ganze Komplexität dieser Verteilung an den Programmierer. Obwohl die sehr große Ausdrucksstärke vorteilhaft erscheint, bringt sie doch auch eine erhöhte Fehleranfälligkeit mit sich [Wikb]. Implizite Parallelität nimmt dem Programmierer diese Bürde ab und macht diese Problematik dadurch transparent. Dadurch wird das Potential der Optimierung durch spezielles Wissen des Programmierers über Abläufe eines Programms natürlich verschenkt [Wikc]. Unter Anderem deshalb sind Umsetzungen wohl in der Regel hybride Lösungen, die dem Programmierer zwar Aufgaben abnehmen, aber dennoch Raum für Optimierungen lassen.

Die Sprache Haskell besitzt nach der Theorie Eigenschaften, die eine gute Eignung für paralleles Programmieren garantieren.

Datenabhängigkeiten erschweren die Erzeugung von parallelen Programmen. Deren Auflösung ist kein triviales Problem, egal ob diese Aufgabe der automatischen Programmtransformation oder dem Programmierer selbst zufällt. Da Haskell eine rein funktionale Sprache ist, treten - abgesehen von monadischen Berechnungen - keine Seiteneffekte auf. Interessanterweise sollte dadurch die Problemklasse Anti-Abhängigkeit, bei der eine Instruktion ein Datum überschreibt, das in einer vorangehenden Instruktion gelesen wird, per Konzept wegfallen.

Im Folgenden wird eine Auswahl an für Haskell vorhandenen Ansätzen nach obiger Kategorisierung eingeordnet und beschrieben.

2 Concurrent Haskell

Der Inhalt dieses Kapitels wurde zum Großteil aus [PJGF96] entnommen.

Concurrent Haskell ist eine konkurrente Erweiterung für Haskell. In Anlehnung an den Einsatz von Monaden für Input/Output (I/O) und gekapselte veränderbare Zustände, ist Concurrent Haskell ein Versuch, die Welten von unübersichtlichen I/O intensiven Programmen und rein funktionalen Sprachen anzunähern.

Concurrent Haskell integriert Konkurrenz in eine lazy-Sprache. D. h. Threads können durch noch nicht ausgewertete Datenstrukturen kommunizieren.

Es gibt in Concurrent Haskell eine Unterteilung in eine deterministische und eine konkurrente Schicht. Existierende Techniken der Beweisführung werden dadurch nicht angetastet. Wenn beispielsweise eine Programmtransformation für ein sequenzielles Programm erwiesenermaßen korrekt ist, so gilt dies auch für Programme von Concurrent Haskell.

2.1 Grundidee

Concurrent Haskell erweitert Haskell um zwei neue Bestandteile:

- Threads und einen Mechanismus um Threads zu erzeugen
- atomar veränderbare Zustände, zur Unterstützung von Thread-Kommunikation und Synchronisation

2.2 Threads

In Concurrent Haskell gibt es die Primitive `forkIO`, die einen konkurrenten Thread startet:

```
1 forkIO :: IO () -> IO ThreadId
```

`forkIO a` ist dabei eine Aktion, die eine andere Aktion `a` als Argument nimmt, und einen neuen Thread erzeugt und startet, der diese Aktion ausführt. Die I/O und andere Seiteneffekte von `a` sind dabei in unbestimmter Weise mit denen, die nach dem forkIO noch folgen überlagert. Betrachten wir folgendes Beispiel:

```
1 let
2  loop ch = hPutChar stdout ch >> loop ch
3 in
4 forkIO (loop 'a') >>
5 loop 'z'
```

`forkIO` erzeugt hier einen Thread, der die Aktion `loop 'a'` ausführt. In der Zwischenzeit führt der Eltern-Thread die Aktion `loop 'z'` aus. Das Resultat ist eine unendlich lange Sequenz von `a`'s und `z`'s. Die genaue Reihenfolge ist unbestimmt.

`forkIO` besitzt folgende Eigenschaften:

- Wegen der Lazyness von Haskell muss die zugrunde liegende Implementierung zwangsläufig die Kommunikation und Synchronisation von Threads unterstützen. Weil nämlich ein Thread einen Ausdruck auswerten könnte, der bereits von einem anderen Thread ausgewertet wird. Dann muss dieser Thread blockiert werden, bis der andere mit der Auswertung fertig ist

- Da Eltern- und Kind-Thread beide den gleichen geteilten Zustand verändern könnten, führt `forkIO` Nichtdeterminismus ein. Wenn beispielsweise ein Thread eine Datei liest und ein anderer die Datei löscht, kann das Ergebnis nicht vorausgesagt werden. Nichtdeterminismus ist nicht wünschenswert aber ist dennoch eine unvermeidbare Eigenschaft von konkurrenten Sprachen [PJGF96].

2.3 Synchronisation und Kommunikation

Zur Kommunikation und Synchronisation von Threads wurde einem zusätzlichen Mechanismus der Vorzug gegenüber den in der Sprache bereits vorhandenen lazy-ausgewerteten Listen gegeben. Gründe dafür sind:

- Threads brauchen möglicherweise exklusiven Zugriff auf Objekte der echten Welt, wie Dateien. Eine Implementierung dieser Eigenschaft greift zwangsläufig auf geteilte, veränderbare Variablen oder Semaphore zurück.

- Wenn ein Thread mit mehreren anderen Threads kommuniziert, so braucht er pro Kommunikationspartner eine eigene Liste. Datenstrukturen wie ein Stream, den mehrere Threads beschreiben oder lesen sollen, sind auf diese Art nur sehr schwer zu implementieren.

Zunächst gibt es einen neuen primitiven Typ

```
1 type MVar a
```

Ein Wert vom Typ `MVar t`, für einen beliebigen Typ `t`, ist der Name einer veränderbaren Speicherposition die entweder leer ist, oder einen Wert vom Typ `t` enthält. Es gibt eine Reihe von primitiven Operationen für `MVar`:

- `newEmptyMVar :: IO (MVar a)` erzeugt ein neues nicht besetztes `MVar`.

- `newMVar :: a -> IO (MVar a)` erzeugt ein neues `MVar`, das initial mit dem angegebenen Wert vom Typ `a` besetzt wird.

- `takeMVar :: MVar a -> IO a` blockiert, bis die Speicherposition nicht-leer ist, gibt dann den enthaltenen Wert zurück und leert die Speicherstelle. Wenn es einen oder mehrere Threads gibt, die durch ein `putMVar` blockiert sind, so wird einer

davon zur Fortsetzung freigegeben, während die anderen weiter warten müssen. Die Reihenfolge wird nach dem first-in-fist-out (FIFO) Prinzip bestimmt, d. h. der Thread, der schon am längsten wartet, wird als erster bedient.

- `putMVar :: MVar a -> a -> IO ()` blockiert, bis die Speicherstelle leer ist, schreibt dann einen Wert in die angegebene Speicherposition. Wenn es einen oder mehrere Threads gibt, die durch ein `takeMVar` blockiert sind, so wird einer davon zur Fortsetzung freigegeben. Die Reihenfolge wird wie bei `takeMVar` nach dem FIFO Prinzip bestimmt.

`MVar` kann auf verschiedene Arten gesehen werden:

- als synchronisierte Version von `MutVar`, bekannt aus monadischem I/O,
- als Channel, mit `takeMVar` und `putMVar` in der Rolle von receive und sen,
- als binäre Semaphore mit den Signal- und Warteoperationen implementiert durch `putMVar` und `takeMVar`.

2.4 Software Transaktionsspeicher

[Har+05]

Konkurrierende Programmierung ist sehr trickreich. Die dominierende Technik basiert auf Sperren, einem einfachen und direkten Ansatz, der allerdings nicht mit Größe und Komplexität eines Programms skaliert. Für die Korrektheit müssen Programmierer herausfinden, welche Operationen im Konflikt stehen. Um Lebendigkeit zu gewährleisten müssen Deadlocks verhindert werden. Für gute Performanz müssen die Granularität von Sperren den dadurch entstehenden Kosten gegenübergestellt werden. Sperren-basierte Programme können nicht ohne Weiteres komponiert werden können. Korrekte Teilprogramme können in Komposition fehlerhaft sein.

Man denke z. B. an eine Hash-Tabelle mit Thread-sicheren Operationen für Einfügen und Löschen. Man möchte jetzt ein Element a aus Tabelle t_1 löschen und in Tabelle t_2 einfügen. Der Zwischenzustand, bei dem das Element a bereits aus t_1 gelöscht, aber noch nicht in t_2 eingefügt ist, sollte jedoch für andere Threads unsichtbar bleiben. Wenn man diese Erweiterungsmöglichkeit bei der Implementierung der Hash-Tabelle nicht vorhergesehen hat, so gibt es keine Möglichkeit, diese Spezifikation in einer Sperren-basierten Umgebung zu erfüllen. Selbst wenn Methoden wie LockTable und UnlockTable eingebaut wurden, können bei falscher Anwendung Deadlocks oder Wettlaufsituationen entstehen. Einfügen und Löschen können somit nicht einfach zu einer Operation komponiert werden.

Eine viel versprechende Alternative zu Sperren bei der Kontrolle der Konkurrenz sind atomare Speichertransaktionen auch bekannt als Transaktionsspeicher. Im obigen Beispiel kann man mit Speichertransaktionen die komponierte Operation auf der Hash-Tabelle ganz einfach durch

```
1 atomic do { v:=delete(t1,A); insert(t2,A,v) }
```

ausdrücken. Diese einfache Konstruktion ist von der Implementierung von Einfügen und Löschen völlig abstrahiert und auch dann korrekt, wenn diese Operationen Sperren enthalten.

2.4.1 Transaktionsspeicher

Transaktionen sind nicht neu: Sie stammen aus dem Datenbank-Design. Die beim Transaktionsspeicher ist dabei, dass ein Code-Block inklusive Verschachtelungen eingeschlossen werden kann in einen atomaren Block. D. h. man hat die Garantie, dass der eingeschlossene Code-Block für alle anderen atomaren Code-Blöcke atomar läuft. Zur Umsetzung von Transaktionsspeicher kann man auf optimistische Synchronisierung zurückgreifen. Statt Sperren anzufordern läuft ein atomarer Block sperrenfrei, akkumuliert aber ein Thread-lokales Transaktionslog, das jeden Schreib- und Lesezugriff auf Speicher aufzeichnet. Wenn ein Block b_1 zum Abschluss kommt, so wird zunächst geprüft, ob seine Sicht auf den Speicher konsistent war und übernimmt dann erst seine Änderungen in den Hauptspeicher. Wenn die Überprüfung fehlschlägt, weil ein anderer Block b_2 in der Zwischenzeit den Hauptspeicher dahingehend verändert hat, dass der gelesene Speicher von b_1 betroffen ist, so wird b_1 neu gestartet.

So eliminiert Transaktionsspeicher per Konstruktion viele Probleme der Sperren-basierten Programmierung.

2.4.2 Umsetzung in Haskell

Haskell ist als rein deklarative Sprache eine gute Basis für Transaktionsspeicher. Zum Einen trennt das Typsystem Berechnungen mit und ohne Seiteneffekten. Dadurch können Transaktionen so gestaltet werden, dass sie zwar den Speicher beeinflussen können, aber keine irreversiblen I/O Operationen durchführen dürfen. Zum Anderen sind Zugriffe auf den geteilten Speicherbereich selten. Der Großteil der Berechnungen ist rein funktional und damit seiteneffektfrei. Somit kann der Großteil der Berechnungen vom Transaktionsspeicher außer Acht gelassen werden. Nur die expliziten Zugriffe auf den gemeinsamen Speicherbereich müssen mitgeschrieben werden. Deshalb ist eine Implementierung auf Software-Ebene für Transaktionsspeicher angemessen.

Transaktionale Variablen und Atomarität

Angenommen man möchte einen Ressourcen-Manager implementieren, der eine bestimmte Anzahl an Ressourcen, repräsentiert durch einen Integer-Wert, verwaltet. Der Aufruf `getR r n` soll `n` Einheiten der Ressource `r` erlangen, blockieren wenn `r` nicht genug Ressourcen hat. Der Aufruf `putR r n` sollte `n` Einheiten der Ressource an `r` zurückgeben. Eine Implementierung von `putR` in Haskell könnte so aussehen:

```
1 type Resource = TVar Int
2 putR :: Resource -> Int -> STM ()
3 putR r i = do { v <- readTVar r
4                ; writeTVar r (v+i) }
```

Die momentan verfügbare Resource wird in der transaktionalen Variable vom Typ `TVar Int` gehalten. Die `type` Deklaration gibt diesem Typ lediglich eine eigene Bezeichnung. Die Funktion `putR` liest den Wert `v` der Ressource aus und schreibt `(v+i)` in dieselbe Zelle zurück.

`readTVar` und `writeTVar` geben beide STM-Aktionen zurück, die durch die `do ...` Notation zu komponierten STM-Aktionen werden können. Diese STM-Aktionen bleiben während ihrer Ausführung vorläufig: Um eine STM-Aktion an das System weiterzugeben, kann es einer Funktion `atomic` mit dem Typ

```
1 atomic :: STM a -> IO a
```

übergeben werden. Diese nimmt eine Speichertransaktion des Typs `STM a` und gibt eine I/O Aktion zurück die bei Ausführung die Speichertransaktion atomar bezüglich aller anderen Speichertransaktionen durchführt. Zusammen ergibt das:

```
1 main = do { ...; atomic (putR r 3); ... }
```

Die Funktion `atomic` und alle Operationen vom Typ `STM` sind über Transaktionsspeicher gebaut. Dadurch wird pro Thread ein Transaktionslog geführt, das die vorläufigen Zugriffe auf `TVar`s mitschreibt. Wenn `atomic` aufgerufen wird, überprüft der STM, ob die mitgeschriebenen Zugriffe valide sind, d. h. keine konkurrierende Transaktion hat dazu im Konflikt stehende Veränderungen am Speicher in das System übernommen. Wenn dies bestätigt wird, werden alle vorläufigen Veränderungen vom STM atomar übernommen und für die anderen Transaktionen sichtbar gemacht. Wenn dies nicht bestätigt wird, so wird die Transaktion mit einem neuen leeren Log neugestartet.

Die Aufteilung in STM-Aktionen und I/O-Aktionen stellt zwei wertvolle Garantien zur Verfügung:

- Nur STM-Aktionen und reine Berechnungen können innerhalb einer Speichertransaktion durchgeführt werden, I/O-Aktionen nicht. Verhindert wird das durch das Typsystem von Haskell. I/O-Aktionen haben nämlich den Typ `IO a` und können deshalb nicht mit STM-Aktionen komponiert werden.

- Außerhalb einer Transaktion können keine STM-Aktionen durchgeführt werden, so dass nicht versehentlich ein `TVar` ohne den Schutz durch `atomic` verändert werden kann.

Blockierende Speichertransaktionen

In jeder Form der konkurrenten Programmierung muss ein Mittel für Threads zur Verfügung stehen, dass den Thread auf Ereignisse, die von anderen Threads induziert werden, warten lässt. Bei Sperren-basierter Programmierung wird dies durch Zustandsvariablen realisiert. Nachrichten-basierte Systeme stellen ein Konstrukt für das Warten auf Nachrichten auf einer bestimmten Anzahl von Kanälen zur Verfügung. Andere STM-Systeme erlauben die bedingte Ausführung eines atomaren Blocks anhand eines boolschen Ausdrucks. Keine dieser Methoden erlaubt eine Komposition atomarer Blöcke.

Die Umsetzung für Haskell bietet einen einfachen Mechanismus, der Sperren und eine Komposition atomarer Blöcke realisiert: Die STM-Aktion `retry :: STM a`. Bezogen auf das obige Beispiel kann nun die Implementierung von `getR` angegeben werden:

```
1 getR :: Resource -> Int -> STM ()
2 getR r i = do { v <- readTVar r
3                ; if (v < i) then retry
4                  else writeTVar r (v-i) }
```

`getR` liest den Wert `v` der Ressource aus und wenn `(v >= i)`, dann wird `v` um `i` verringert. Wenn `(v >= i)` jedoch nicht gilt, d. h. es sind nicht genug Einheiten der Ressource vorhanden, dann wird `retry` aufgerufen. `retry` bricht die Transaktion ab ohne eine Veränderung im System zu hinterlassen und startet sie neu. Natürlich gibt es keinen Grund die Transaktion tatsächlich neuzustarten, bevor nicht zumindest eine der während der gescheiterten Transaktion gelesenen `TVar`s durch eine andere Transaktion verändert wurde. Im Transaktionslog stehen die Leseoperationen auf genau diese `TVar`s. Der ausführende Thread wird nun so lange blockiert, bis mindestens eine dieser `TVar`s verändert wird. Der Typ von `retry` ist `STM a`, d. h. es kann immer innerhalb einer STM-Aktion aufgerufen werden.

Im Gegensatz zur Prüfung auf Validität vor dem Abschluss einer Transaktion, ist `retry` nicht automatisch und implizit, sondern explizit. `retry` ist dabei kein Indikator für unerwartetes Verhalten oder Fehler in einer Transaktion, sondern ein Ersatz für explizite Blockaden aus der Sperren-basierten Programmierung.

In `putR` gibt es auch keine Signale von Zustandsvariablen. Dieses Signal wird implizit durch eine Veränderung an den `TVar`s induziert. Die Fehlerklasse lost-wake-up wird damit vollständig beseitigt.

Sequentielle Komposition

Durch den Einsatz von `atomic` identifiziert man atomare Transaktionen im Sinne von: Alle enthaltenen Aktionen scheinen unteilbar ausgeführt zu werden. Das ist der Schlüssel zu sequenzieller Komposition. Z. B. kann man das Einsammeln von drei Einheiten einer Resource und sieben einer anderen ausdrücken durch:

```
1 atomic (do { getR r1 3; getR r2 7 })
```

Die Notation `do ... ; ...` komponiert die STM-Aktionen der beiden `getR` Aufrufe und der zugrundeliegende Transaktionsspeicher übernimmt die Veränderungen beider Aktionen atomar in das System.

`retry` ist das zentrale Element, wenn es darum geht STM-Aktionen, die blockieren können, zu komponieren. Die oben angegebene Funktion wird blockieren wenn entweder `r1` oder `r2` nicht genügend Ressourcen haben. Für den Aufrufer ist es belanglos, wie `getR` implementiert ist oder unter welcher Bedingung der Aufruf erfolgreich ist. Außerdem gibt es kein Risiko, einen Deadlock zu erzeugen, weil auf `r2` gewartet werden muss, während eine Sperre auf `r1` gehalten wird. Die Kapselung als STM-Aktion ist verantwortlich für die Kompositionalität von Transaktionen. Erst nach der Komposition wird durch `atomic` die Transaktion verpackt und ausführbar gemacht.

Komposition von Alternativen

STM Haskell bietet die Möglichkeit, Transaktionen als Alternativen zu Komponieren, so dass nur eine davon ausgeführt wird. Um beispielsweise entweder drei Einheiten von `r1` oder sieben Einheiten vo `r2` zu bekommen, schreibt man:

```
1 atomic (do { getR r1 3 `orElse` getR r2 7 })
```

Die Transaktion `s1 `orElse` s2` führt zunächst `s1` aus. Wenn es zu einem `retry` kommt, so wird `s1` effektlos abgebrochen und `s2` ausgeführt. Wenn es bei `s2` ebenfalls zu einem `retry` kommt, so wird die ganze komponierte Transaktion abgebrochen und neugestartet. Allerdings wird in dem Fall neugestartet, sobald die gelesenen Variablen von `s1` oder `s2` verändert wurden. Der Programmierer muss wiederum nichts über die Bedingung zur Wiederausführung von `s1` oder `s2` wissen.

Der Einsatz von `orElse` bietet eine elegante Möglichkeit, die Entscheidung, ob ein Aufruf an eine Funktion einer Bibliothek blockierend oder nicht blockierend sein soll, dem Aufrufer selbst zu überlassen. Die blockierende Funktion `getR` kann beispielsweise folgendermaßen in eine nicht-blockierende Funktion transformiert werden, die eine Boolsche Variable als Erfolgs-Indikator zurückgibt:

```
1 nonBlockGetR :: Resource -> Int -> STM Bool
2 nonBlockGetR r i = do { getR r i ; return True }
3                       `orElse` return False
```

`nonBlockGetR` kann auf diese Weise ausgedrückt werden, weil `orElse` so implementiert ist, dass erst das erste Argument und dann das zweite Argument ausgewertet wird. Das gleiche Konstrukt kann genauso dafür verwendet werden, um eine blockierende Transaktion aus einer nicht-blockierenden zu erzeugen, die einen Boolschen Wert zurückgibt.

```
1 blockGetR :: Resource -> Int -> STM ()
2 blockGetR r i =
3        do { s <- nonBlockGetR r i;
4             if s then return () else retry }
```

`orElse` besitzt weitere nützliche Eigenschaften: es ist assoziativ und hat `retry` als neutrales Element

```
1 M1 `orElse` (M2 `orElse` M3)
2                   = (M1 `orElse` M2) `orElse` M3
3 retry `orElse` M = M
4 M `orElse` retry = M
```

2.4.3 Performanz

In einer praktischen Studie zu Haskell STM in [Dis+06] wurde eine bestehende konkurrente Bibliothek zum Teil in Haskell auf zwei verschiedene Arten neu implementiert: Erst unter Verwendung expliziter Sperren und dann mit Hilfe von STM.

Zur Untersuchung der Performanz wurden beide Implementierungen herangezogen. Genauer wurden die Implementierungen bei variabler Anzahl an Prozessorkernen eingesetzt,

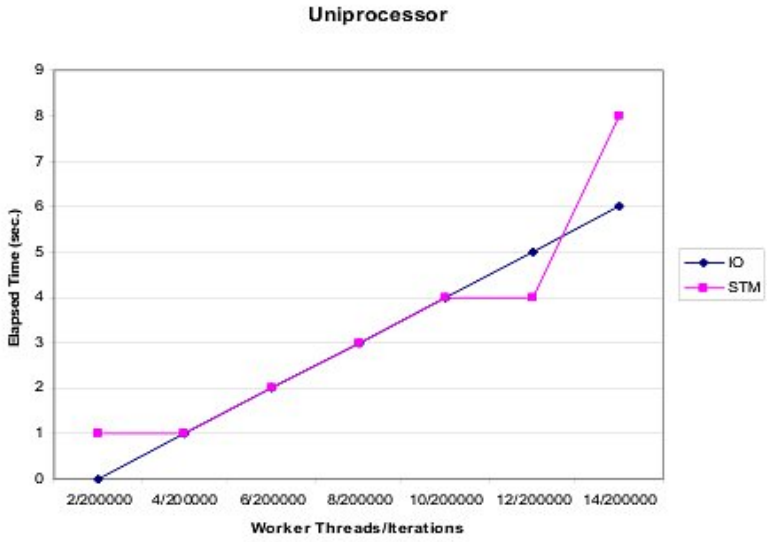

Abbildung 2.1: Performanz mit einem Prozessor. [Dis+06]

um herauszufinden, wie viel Parallelismus von jedem Ansatz ausgenutzt werden kann. Die Ergebnisse sind jedoch nur vorläufig, weil die Implementierung noch unausgereift ist.

Genauer wird in den Tests eine ArrayBlockingQueue vom Typ Integer erzeugt und von einer variablen Zahl an Schreib- und Lese-Threads darauf zugegriffen. Die Schreib und Lese-Threads arbeiten eine feste Anzahl an Operationen ab, gemessen wird die Laufzeit des gesamten Programms. Die Resultate (siehe Abbildung 2.1, 2.2 und 2.3) sind ermutigend. Auf einem Prozessor waren die Resultate von Sperren- und STM-basiertem Programm nahezu identisch. Auf mehreren Prozessoren war die STM-Version konstant schneller als die Sperren-basierte.

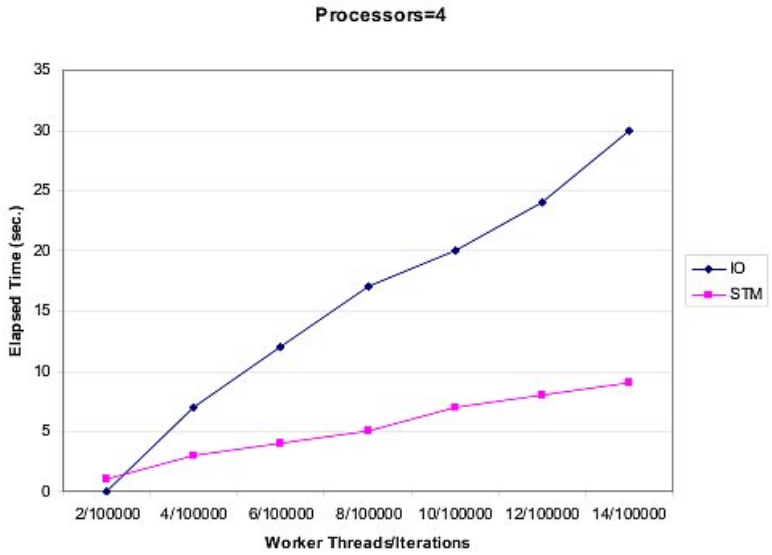

Abbildung 2.2: Performanz mit vier Prozessoren. [Dis+06]

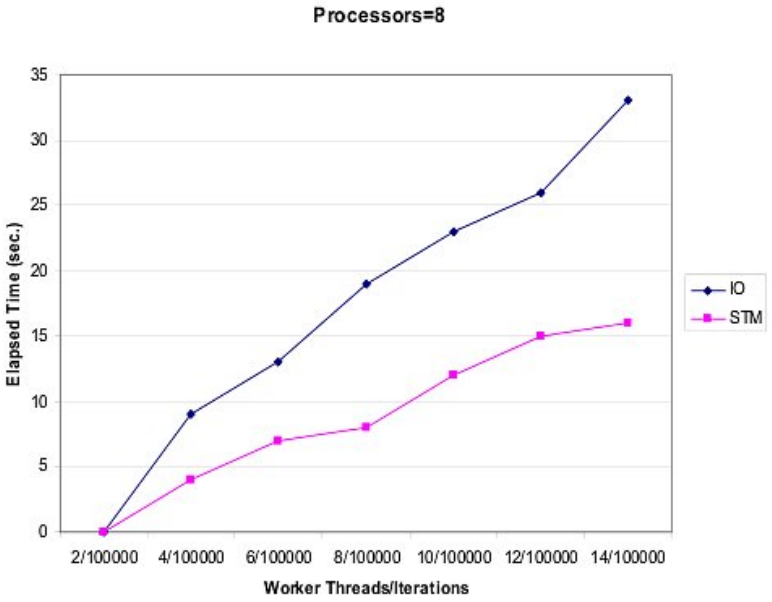

Abbildung 2.3: Performanz mit acht Prozessoren. [Dis+06]

3 Glasgow Parallel Haskell

Der Inhalt des folgenden Kapitels stammt zum Großteil aus [Mar+10].

3.1 Primitiven

Parallelismus wird in Haskell erreicht durch die Primitiven `par` und `pseq` mit den Typen:

```
1 par  :: a -> b -> b
2 pseq :: a -> b -> b
```

Der `par` Kombinator führt zu einer potentiell parallelen Auswertung. Angewendet auf zwei Argumente `p` und `q` wird der Wert von `q` zurückgegeben, während p möglicherweise parallel dazu ausgewertet wird. Nur möglicherweise, weil das Resultat von `par a b` immer `b` ist. Es ergibt für die Bedeutung des Programms keinen Unterschied, ob `a` parallel ausgewertet wird oder nicht. Man sollte `par` als Annotation betrachten. Es weißt die zugrundeliegenden Haskell Implementierung darauf hin, dass es günstig sein könnte, das erste Argument parallel auszuwerten.

Selbst wenn der parallel ausgewertete Ausdruck den Wert $\perp$ ergibt, muss dass Laufzeitsystem dafür sorgen, dass `par a b` semantisch immer `b` ergibt. Für typische Haskell Implementierungen ist dies kein Problem, weil eine lazy-Auswertung auch den Wert $\perp$ haben kann.

`par` alleine ist nicht ausreichend, weil man bei einer parallelen Auswertung bestimmen können sollte, mit welcher anderen Auswertung die Auswertung parallel stattfinden soll. In Haskell wird eine bestimmte Auswertungsreihenfolge weder spezifiziert noch benötigt, deshalb hat der Programmierer in der Regel keinen Einfluss auf diesen Aspekt der Programmausführung. Man hat keine Kontrolle darüber, wann ein bestimmter Aufruf von par ausgewertet wird, oder was davor bzw. dannach ausgewertet wird. Dafür gibt es `pseq`: Ein Aufruf `pseq a b` spezifiziert eine Anforderung an die Auswertungsreihenfolge, so dass `a` nach weak-head normal form (WHNF) ausgewertet wird, bevor b evaluiert zurückgegeben wird. Die denotationelle Semantik von `pseq` ist

```
1 pseq a b = ⊥, if a = ⊥
2          = b,       otherwise
```

und die operationale Semantik ist, dass `a` nach WHNF evaluiert werden muss, bevor `b` evaluiert wird.

Ein Beispiel zum Einsatz von `par` und `pseq` bei der Berechnung der Fibonacci Funktion:

```
1 fib :: Int -> Int
2 fib n
3   | n <= 1    = 1
```

```
4     | otherwise = let
5                         x = fib (n-1)
6                         y = fib (n-2)
7                     in
8                         x `par` y `pseq` x + y
```

Die Fibonacci Berechnung ist aufgebaut wie ein Binärbaum. An jedem Knoten der Berechnung werden `par` und `pseq` kombiniert, um einen Zweig parallel mit dem anderen Zweig auszuwerten. Das Muster ist gebräuchlich: in `x `par` (y `pseq` e)` bezieht `e` typischerweise `x` und `y` mit ein. Der Effekt dieses Musters ist, dass `x` und `y` parallel ausgewertet werden. Wenn die Auswertung von `y` abgeschlossen ist, wird mit der Auswertung von `e` fortgefahren. `pseq` wird hier zur Bestimmung der Auswertungsreihenfolge eingesetzt.

Die Parallelität ist hier unabhängig von der Anzahl der Prozessoren: Jedes mal, wenn `par` evaluiert wird, wird eine neue Möglichkeit generiert, einen Ausdruck parallel auszuwerten (ein *spark*), der Implementierung ist es jedoch freigestellt, diese Möglichkeit zu ignorieren. Typischerweise werden durch den Einsatz von `par` viel mehr sparks erzeugt, als Prozessoren zur Auswertung vorhanden sind und die Überschüssigen sparks werden verworfen.

3.2 Ursprüngliche Strategien

Das oben beschriebene Programmiermodell bietet eine Basis um Parallelität auszudrücken. Aufbauend darauf, wurden Strategien als Abstraktionsschicht über `par` und `pseq` eingeführt, um größer dimensionierte parallele Algorithmen auszudrücken. In der ursprünglichen Formulierung ist eine Strategie eine Funktion vom Typ `a -> ()` für beliebiege `a`.

```
1 type Strategy a = a -> ()
```

Somit kann eine Strategie ihr Argument entweder ganz oder teilweise evaluieren und nur `()` zurückgeben oder divergieren. Insbesondere kann eine Strategie durch den Einsatz von `par` und `pseq` ein Muster enthalten, wie ihr Argument parallel ausgewertet werden kann.

Grundlegende Strategien können wie folgt definiert werden:

```
1 r0 :: Strategy a
2 r0 x = ()
3
4 rwhnf :: Strategy a
5 rwhnf x = x `pseq` ()
6
7 rnf :: NFData a => Strategy a
```

`r0` ist eine Strategie, die nichts von ihrem Argument evaluiert, `rwhnf` wertet ihr Argument nach WHNF aus und `rnf` komplett, d. h. nach Normalform. Die Definition von `rnf` ist abhängig von der Struktur des Arguments. Deshalb ist es so definiert, dass die Typklasse `NFData` verwendet wird, die für jeden Datentyp separat implementiert werden muss. Für

gebräuchliche Datentypen wie Boolsche Variablen, Integer, Listen und Tupel ist eine Implementierung in der Strategien-Bibliothek bereits vorhanden.

Strategien werden mit dem Operator `using` angewendet:

```
1 using :: a -> Strategy a -> a
2 using x s = s x `pseq` x
```

Eine einfache Strategie, die tatsächliche Parallelität enthält ist `parList`. Sie wendet eine Strategie parallel auf jedes Listenelement an.

```
1 parList :: Strategy a -> Strategy [a]
2 parList strat []     = ()
3 parList strat (x:xs) = strat x `par`
4                        parList strat xs
```

Die Funktion `parList` illustriert die kompositionelle Eigenschaft der Abstratktion durch Strategien: sie nimmt als Argument eine Strategie, die auf jedes Element der Liste angewendet werden soll und gibt eine Strategie für die ganze Liste zurück. Das Argument wird dabei typischerweise zur Spezifizierung des Grades der Auswertung verwendet. `parList rwhnf` beispielsweise lässt jeden Spark sein Listenelement bis zum Top-Level Konstruktor auswerten, wohingegen `parList rnf` jedes Listenelement komplett auswertet. An `parList` wird auch die modulare Natur der Strategien sichtbar:

```
1 parMap strat f xs = map f xs `using` parList strat
```

Die `parMap` Funktion nimmt eine Strategie `strat`, eine Funktion `f`, eine Liste `xs` als Argumente und wendet die Funktion `f` parallel auf jedes Listenelement an unter Verwendung von `strat`. Man sieht, wie die der Code zur Parallelisierung rechts durch `using` vom Rest entkoppelt wird.

Der Schlüssel zur Modularität steckt in der lazy-Auswertung. Das Argument einer Strategie kann eine komplexe Datenstruktur mit lazy Komponenten sein, oder sogar eine lazy-erzeugte Datenstruktur, was den Algorithmus, der die Datenstruktur erzeugt, von der Evaluierungsstrategie trennt. Trotzdem sind Strategien kein Allheilmittel: nicht alle Algorithmen lassen sich auf diese Weise zerlegen und die erzeugte lazy Zwischen-Datenstruktur bringt auch Kosten mit sich. Trotzdem kann in vielen Fällen die gewonnene Modularität die Kosten überwiegen und manchmal kann die Zwischen-Datenstruktur durch den Compiler automatisch wegoptimiert werden.

3.3 Speicherverwaltung

Wenn in einem Haskell Programm der Ausruck `par a b` ausgewertet wird, speichert das Laufzeitsystem einen Zeiger auf den Heap-Knoten, der `a` in einer Datenstruktur namens Spark-Pool repräsentiert. In dieser Sichtweise besteht der Spark-Pool lediglich aus einer Menge von Zeigern auf Heap-Objekte, die Berechnungen repräsentieren, die durch `par` gesparked wurden. Das Laufzeitsystem entfernt von Zeit zu Zeit Objekte aus dem Pool, um sie auf unausgelasteten Prozessoren auszuwerten.

Es gibt in der Hauptsache zwei Richtlinien, wie die Speicherverwaltung, genauer gesagt der Garbage-Collector den Spark-Pool behandeln kann:

- ROOT: Einträge des Spark-Pool sollten als implizit lebendig betrachtet werden. D. h. der Spark-Pool ist eine Quelle von Wurzeln für den Garbage-Collector.
- WEAK: Ein Eintrag des Spark-Pools ist nur lebendig, wenn das Objekt, auf das er zeigt, unabhängig erreichbar ist. Das bedeutet, dass der Spark-Pool schwache Zeiger enthält.

Tatsächlich führen beide Richtlinien zu Problemen bei den ursprünglichen Strategien. Zunächst betrachten wir die WEAK-Richtlinie und untersuchen, wie es mit der o. g. Definition von `parList` umgeht. Die Sparks, die von `parList` erzeugt werden haben alle die Form `(strat a)` für eine Strategie `strat` angewendet auf ein Listenelement `a`. Jeder dieser Ausdrücke wird nur angelegt, damit er an `par` übergeben werden kann. Der Spark-Pool wird Referenzen auf viele Ausdrücke der Form `(strat x)` enthalten und in jedem Fall ist diese Referenz auch die einzige auf diesen Ausdruck auf dem Heap. Deshalb würde der Garbage-Collector bei Anwendung der WEAK-Richtlinie jeden so erzeugten Spark entfernen was zwangsläufig zu einem Verlust der Parallelität führte.

Darüberhinaus gibt es keine Definition von `parList`, die dieses Problem umgehen könnte. `()` ist der einzigen Wert, den die `parList` Strategie zurückgeben kann, deshalb wäre die einzige Möglichkeit einen vom Spark-Pool unabhängig erreichbaren Spark zu erzeugen, einen Teil der ursprünglichen Struktur selbst zu sparken, so wie die Elemente der Liste. Das sähe dann so aus:

```
1 parListWHNF :: Strategy [a]
2 parListWHNF []     = ()
3 parListWHNF (x:xs) = x `par` parListWHNF strat xs
```

Leider ginge dabei die kompositionelle Eigenschaft der Strategien verloren.

Bei der Betrachtung von `parList` unter der ROOT-Richtlinie würde der Spark-Pool nach wie vor die Referenzen auf Ausdrücke der Form `(strat x)` enthalten, aber in diesem Fall würden diese nicht von Garbage-Collector gelöscht, d. h. die Parallelität bliebe erhalten. Allerdings entsteht dadurch ein anderes Problem: Wenn nämlich nicht ausreichend Prozessoren vorhanden sind, um alle Sparks auszuwerten, behält der Spark-Pool die Referenzen auf alle Ausrücke `(strat x)`, möglicherweise auch wenn jedes `x` vom Programm längst nicht mehr benötigt wird.

Zur Erhaltung der Parallelität hält die Speicherverwaltung so den Speicher zurück, der längst hätte freigeben werden sollen: Das bildet ein Speicherleck und kann zu starken Einbußen bei der Performanz führen. Ein harmloses `parList` oder `parMap` kann ein Programm von konstantem Speicherverbrauch zu linearem Speicherverbrauch bringen. Auch wenn diese nachteiligen Effekte hauptsächlich bei der Programmausführung mit nur einem Prozessor auftreten, weil dann keine freien Prozessoren vorhanden sind, um Sparks auszuwerten und sie so aus dem Spark-Pool zu entfernen, so sind sie auch beim Einsatz mehrere Prozessoren spürbar: die überholten Sparks belegen Speicher im Spark-Pool und Prozessoren verschwenden Zeit darauf, die überholten Sparks auch noch auszuwerten.

3.3.1 Versandete Sparks

Möglicherweise zeigt ein Spark im Spark-Pool auf eine Berechnung, die bereits durch das Programm ausgewertet wurde. Vielleicht waren nicht ausreichend Prozessoren vorhanden, um den Spark parallel auszuwerten und eine anderer Thread hat ihn letztendlich im Verlauf seiner Berechnungen evaluiert.

Wenn ein Spark aus dem Spark-Pool auf einen Wert, statt einer unausgewerteten Berechnung, zeigt, so ist der Spark versandet (aus dem Englischen: fizzled), das Potential für eine parallele Auswertung ist verschwunden. Das Laufzeitsystem kann und sollte versandete Sparks finden und aus dem Spark-Pool entfernen, so dass die Speicherverwaltung den Speicher freigeben kann und keine Zeit auf deren sinnfreie Auswertung verschwendet wird.

In der ursprünglichen Formulierung der Strategien können die meisten Sparks niemals versanden, weil sie aus Ausdrücken der Form `(strat x)` bestehen, die mit dem Hauptprogramm nicht in Verbindung stehen und deshalb von diesem auch nicht ausgewertet werden können. Die einfache nicht-komponierbare Operation `parListWHNF` dagegen generiert Sparks, die versanden können. In ihrem Fall wird `par` direkt auf einen Teil der Datenstruktur selbst, anstatt auf einen neuen Ausdruck, der im Hauptprogramm nicht vorkommt, angewendet. Deshalb wird das Hauptprogramm zwangsläufig irgendwann die Datenstruktur auswerten.

3.3.2 Spekulative Parallelität

Das Erzeugen von Sparks sollte auch spekulative Parallelität unterstützen, d. h. Sparks die Ausdücke enthalten, bei denen es ungewiss ist, ob sie im Verlauf der Auswertung des Hauptprogramms benötigt werden. Idealerweise sollten solche Sparks automatisch verworfen werden, wenn feststeht, dass sie nicht benötigt werden.

Unter Verwendung der ROOT-Richtlinie wird so ein Spark nie verworfen, und dadurch zu einem Speicherleck, wohingegen bei der WEAK-Richtlinie unerreichbare Sparks entfernt werden.

3.4 Neue Strategien

Die Schwierigkeiten bei der Handhabung des Speicherverhaltens von Sparks liegen bei dem Typen der Strategie-Funktionen: Wenn eine Strategie-Funktion den Typ Unit `()` zurückliefert, gibt es keine Möglichkeit einen Spark mit einem bestimmten Ausdruck zu erzeugen und den Ausdruck an den Aufrufer zurückzugeben, um so den Ausruck in dem Spark für den Garbage-Collector greifbar zu machen.

Die Hauptidee bei der Neuformulierung der Strategien ist, dass eine Strategie eine *neue Version* ihres Arguments zurückgibt, in die die Berechnungen aus einem Spark eingebettet sind. Beispielsweise soll die Strategie bei der Evaluierung eines Sparks mit dem Ausdruck `(strat x)` das Ergebnis nicht verwerfen, sondern eine neue Version der Datenstruktur erzeugen, so dass `strat x` anstelle von `x` steht. Der Aufrufer wird die neue Datenstruktur konsumieren und die alte verwerfen, so dass der Spark `(strat x)`

erreichbar bleibt, so lange der Konsument ihn benötigt. Desweiteren versandet der Spark, wenn der Konsument den Ausdruck des Sparks vor der parallelen Auswertung evaluiert wird. Überflüssige Sparks werden so durch den Garbage-Collector verworfen.

Identitätsfunktionen kommen nicht in Frage, weil die einfachste davon (`a -> a`) strikt ist, weshalb die Strategie `r0`, die keine Auswertung vornimmt, nicht ausgedrückt werden könnte. Um `r0` unterzubringen wird *lifting* angewandt. Dadurch erhalten wir `Eval` und bieten die Möglichkeit zur Umkehrung des lifting: `runEval`

```
type Strategy a = a -> Eval a

data Eval a = Done a

runEval :: Eval a -> a
runEval (Done a) = a
```

Jetzt können die grundlegenden Kombinatoren für Strategien festgelegt werden:

```
r0 :: Strategy a
r0 x = Done x

rseq :: Strategy a
rseq x = x `pseq` Done x

rpar :: Strategy a
rpar = x `par` Done x

rdeepseq :: NFData a => Strategy a
rdeepseq = rnf x `pseq` Done x
```

`r0`, `rseq` und `rdeepseq` entsprechen den ursprünglichen `r0`, `rwhnf` und `rnf`.

3.4.1 Die Auswertungsreihenfolgen-Monade

`Eval` wird als strikte Identitätsmonade deklariert:

```
instance Monad Eval where
  return x = Done x
  Done x >>= k = kx
```

Die strikte Identitätsmonade stellt eine praktische und flexible Notation zum Ausdruck der Auswertungsreihenfolge, d.h der Ordnung zwischen Anwendungen von `rseq` und `rpar`, zur Verfügung. Damit ist sie zum Ausdruck grundsätzlicher Parallelität gut geeignet. Die folgende Definition der Fibonacci-Funktion

```
let
  x = fib (n-1)
  y = fib (n-2)
in
  x `par` y `pseq` x + y
```

kann dadurch umgeschrieben werden als

```
runEval $ do
  x <- rpar (fib (n-1))
  y <- rseq (fib (n-2))
  return x + y
```

was die Ordnung zwischen `rpar` und `rseq` klar zum Ausdruck bringt und eine Notation einsetzt, die Haskell-Programmierer vertraut ist.

Beim Einsatz der neuen Strategien-API werden `par` und `pseq` zur Erzeugung neuer Strategien nicht mehr gebraucht. Stattdessen wird die `Eval`-Monade mit `rpar` und `rseq` eingesetzt. Die `Eval`-Monade erhöht den Grad der Abstraktion für `pseq` und `par`: Fragmente der Auswertungsreihenfolge werden dadurch first-class und komponierbar. Man sollte sich die `Eval`-Monade als eingebettete Domänenspezifische Sprache (aus dem Englischen: Embedded Domain-Specific Language (EDSL)) zum Ausdruck vom Auswertungsreihenfolge vorstellen, die in Haskell, einer Sprache ohne festgelegte Auswertungsreihenfolge eingebettet wird.

3.4.2 Eval, applikativ

Eine Auswertungsreihenfolge ist oft etwas, dass wir einem bestehenden Ausdruck auferlegen möchten. `Eval` ist als Monade auch ein applikativer Funktor:

```
instance Functor Eval where
  fmap f x = x >>= return . f

instance Applicative Eval where
  pure x = return x
  (<*>) = ap
```

Das bedeutet, dass wir die Auswertungsreihenfolge durch applikative Notation ausdrücken können. In folgendem Beispiel ist ein wert `res` definiert als

```
res = append left right
```

und wir wollen `left` und `right` parallel ausführen. Denkbar wäre der Einsatz der monadischen Notation, wie in dem Fibonacci-Beispiel von oben, aber manchmal ist diese Notation zu schwerfällig und verzerrt die Struktur des Ausgangscodes. Mit Hilfe der applikativen Operatoren `pure`, `<$>` und `<*>` können wir den Code parallelisieren, ohne die ursprüngliche Struktur zu verwerfen:

```
res = runEval $ append <$> rpar left <*> rseq right
```

Das ist an dieser Stelle natürlich kein modularer Ansatz zum Ausdruck der Parallelität, aber dennoch kann hier durch einen minimalen, präzisen Eingriff, bestehender Code parallelisiert werden.

3.4.3 Strategien im Einsatz

Genau wie bei den ursprünglichen Strategien, gibt es wiederum einen Operator zur Anwendung der Strategien:

```
using :: a -> Strategy a -> a
x `using` s = runEval (s x)
```

`using` hat die niedrigste Priorität bei der Auswertung und ist linksassoziativ, d. h. `e `using` s1 `using` s2` ist dasselbe wie `(e `using` s1) `using` s2`. Das Stapeln von Strategien auf diese Weise ist dem Stapeln von Funktionen ähnlich. Deshalb gibt es einen Kompositionsoperator `dot`:

```
(e `using` s1) `using` s2 = e `using` (s1 `dot` s2)
```

Wie die Funktionskomposition hat `dot` die höchste Priorität bei der Auswertung und ist rechtsassoziativ.

3.4.4 Komponierte Strategien

Für Strategien über Datentypen sollte zunächst eine grundsätzliche Strategie festgelegt werden, die durch durch Strategien für die Komponenten des Datentyps parametrisiert wird. Diese grundsätzliche Strategie traversiert den Datentyp in der `Eval`-Monade, wendet die per Parameter übergebenen Strategien an und erzeugt eine neue Instanz des Datentyps.

Der Kombinator `evalList` läuft beispielsweise über eine Liste und wendet die übergebene Strategie auf jedes Element an:

```
evalList :: Strategy a -> Strategy [a]
evalList s []     = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

`parList` kann durch eine Komposition der Strategie für die Elemente und `rpar` erzeugt werden:

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Die ursprüngliche Implementierung der Strategien besaß eine entsprechende Funktion `seqList` die mit Hilfe von `pseq` implementiert worden war. `parList` kommt völlig ohne `pseq` aus, die Auswertungsreihenfolge wird durch die `Eval`-Monade explizit festgelegt.

3.4.5 Generische Strategien

Die Klasse `Traversable` bietet eine praktische Umgebung, um `Applicative`-Berechnungen über einer Datenstruktur mit anschließender Erzeugung einer neuen Datenstruktur durchzuführen. Um Strategien über regulären Datenstrukturen, wie Listen oder Bäumen zu definieren ist dies das Mittel der Wahl.

Die Methode `traverse` hat den folgenden Typ:

```
traverse :: (Traversable t, Applicative f)
          => (a -> f b) -> t a -> f (t b)
```

Angepasst an unsere Bedürfnisse durch eine Spezialisierung von `a -> f b` nach `Strategy a`, ergibt das:

```
evalTraversable :: Traversable t
                => Strategy a
                -> Strategy (t a)

evalTraversable = traverse
```

Es handelt sich hierbei nun um eine generisch parametrisierte `Strategy` für jegliche `Traversable`-Datentypen. `evalList` ist eine Instanz davon und Strategien für Typen wie `Maybe` und `Array` sind auch enthalten. Die Parallelisierung der generischen Strategie ist unkompliziert:

```
parTraversable :: Traversable t
               => Strategy a
               -> Strategy (t a)

parTraversable s = evalTraversable (rpar `dot` s)
```

3.4.6 Modularität

Das Schlüsselfeature Modularität bedeutet, dass `e `using` s` äquivalent ist zu `e`, damit man zum Verständnis des bloßen Algorithmus Letzteres einfach weglassen kann. Das bringt natürlich nur etwas, wenn `using` tatsächlich eingesetzt wird. Bei einigen Beispielen war dies nicht der Fall, unter anderem bei der Fibonacci-Funktion:

```
runEval $ do
  x <- rpar (fib (n-1))
  y <- rseq (fib (n-2))
  return x + y
```

Diese Art der Parallelität wird auch Kontroll oder Task-Parallelität genannt, wonach die Parallelität der Kontrollstruktur des Programms folgt. Dies ist keine modulare Spezifikation von Parallelität, weil der Algorithmus mit der Koordination vermischt wird.

Man kann allerdings eine modulare Version schreiben:

```
x + y `using` strat
where
  x = fib (n-1)
  y = fib (n-2)
  strat v = runEval $ do rpar x; rseq y; return v
```

Diese Strategie mag eigenartig erscheinen, weil der Rückgabewert von `rpar` einfach verworfen wird, und dadurch ja eigentlich der Garbage-Collector diesen Spark entfernen sollte, oder daraus ein Speicherleck entsteht. Trotzdem ist diese Lösung korrekt, da das Argument von `rpar` eine Variable ist, die auch im Ergebnis vorkommt.

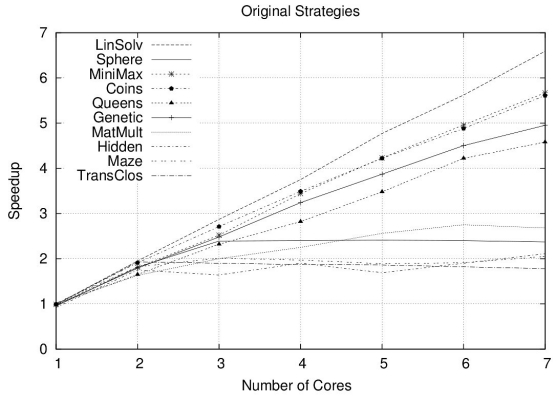

Abbildung 3.1: Speedup bei alten Strategien. [Mar+10]

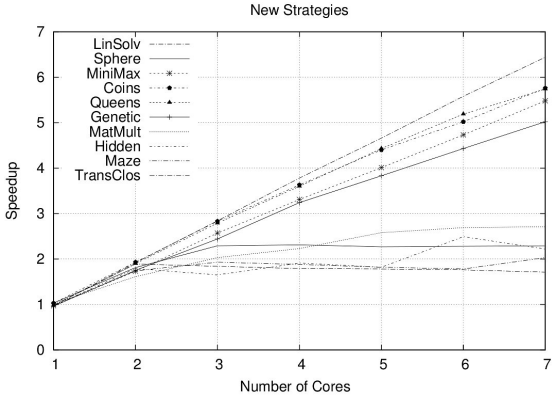

Abbildung 3.2: Speedup bei neuen Strategien. [Mar+10]

3.5 Performanz

Für die Tests wurden die Implementierungen von 10 realitätsnahen Anwendungen, die viele Paradigmen des Parallelprogrammierens abdecken. Dabei wurden die Implementierungen mit den ursprünglichen Strategien (Abbildung 3.1) verglichen mit den Implementierungen mit den neuen Strategien (Abbildung 3.2). Es lassen sich nur marginale Unterschiede entdecken: Bei 7 Kernen liegt der Speedup im arithmetischen Mittel bei den neuen Strategien bei einem Wert von 3,85, während die ursprünglichen Strategien bei 3,72 liegen.

Der Speicherverbrauch bei einzelnen Anwendungen konnte erheblich reduziert werden, da die neuen Strategien Probleme bei der Speicherverwaltung beseitigen.

4 Data Parallel Haskell

Der Inhalt des nachfolgenden Kapitels ist in großen Teilen aus [PJS09; PJ+08] entnommen.

Ein vielversprechender Ansatz, um eine große Anzahl an Prozessoren auszulasten, ist Datenparallelität. Data Parallel Haskell (DPH) verbindet die Flexibiliät von verschachtelter Datenparallelität mit der Schnelligkeit und Skalierbarkeit von flacher Datenparallelität.

4.1 Datenparallelität

In einem Multiprozessor-System wird Datenparallelität bei einzer Menge von Operationen dadurch erzeugt, dass jeder Prozessor dieselbe Aufgabe auf verschiedenen Teilen von verteilten Daten ausführt. Diese Architektur wird in Flynns Taxonomie auch als SIMD (Single Instruction, Multiple Data streams) oder MIMD (Multiple Instructions, Multiple Data streams) bezeichnet [Wika].

4.1.1 Flache Datenparallelität

Flache Datenparallelität bedeutet, dass eine Funktion parallel auf eine Menge von Werten angewendet werden kann, während die Funktion an sich sequentiell sein muss [Ble96]. Dabei entstehen:

- gute Granularität: ein grob-granularer Thread pro Prozessor,
- gute Lastverteilung: jeder Prozessor leistet annähernd dasselbe,
- gute Daten-Lokalität: die Daten werden der Reihe nach bearbeitet.

Flache Datenparallelität lässt sich sehr gut auf aktuelle Architekturen, wie SIMD, MIMD, GPGPU oder Multicore abbilden. Der Haken dabei ist, dass nur ein kleiner Teilbereich von Programmen in dieses Paradigma passt und so geschriebene Programme sich nur sehr schlecht bis garnicht komponieren lassen.

4.1.2 Verschachtelte Datenparallelität

Verschachtelte Datenparallelität bedeutet, dass jedwede Funktion, insbesondere parallele, auf einer Menge von Daten angewendet werden kann. Dadurch können auch komplexere Architekturen, wie verschachtelte Schleifen oder Divide-and-Conquer umgesetzt werden [Ble96].

D. h. flache Datenparallelität ist sehr gut automatisch parallelisierbar, während verschachtelte Datenparallelität gut skalierbar, komponierbar und verständlich ist. Verschachtelte Datenparallelität kann durch eine Programmtransformation in flache Datenparallelität überführt werden, was in DPH ausgenutzt wird. Wie diese Transformation im Einzelnen abläuft, wird in Abschnitt 4.3 skizziert. Zunächst betrachten wir jedoch das Programmiermodell von DPH.

4.2 Programmieren mit Data Parallel Haskell

Data Parallel Haskell ist Haskell mit den folgenden zusätzlichen Features:

- Dem Typ *Parallel Array*, notiert als `[:e:]` für Arrays vom Typ `e`. Diese Arrays werden von Werten vom Typ `Int` indexiert. Semantisch betrachtet ist ein Arrray `[:a:]` einer Liste `[a]` sehr ähnlich. Ein Array kann elemente jeglicher Art enthalten, Arrays und Funktionen eingeschlossen.
- Parallele Operationen, die auf parallelen Arrays operieren. Soweit es geht, tragen diese Operationen dieselben Namen wie die entsprechenden Operationen der Liste, nur mit dem Suffix `P`, z. B. `mapP`.
- Syntaktischem Zucker: Parallel Array Comprehensions, ähnlich der list Comprehension, operieren auf Parallel Arrays

Zusätzlich unterscheiden sich Parallel Arrays von Listen in der Striktheit: Wenn ein Element des Parallel Arrays abgefragt wird, so wird jedes einzelne Element ausgewertet.

4.2.1 Beispiel: Wortsuche

Eine minimale Version einer Suchmaschine: Ein `Document` ist ein Array von Wörtern (Strings). Die Aufgabe ist es, alle Vorkommen eines Wortes in einer großen Sammlung von Dokumenten zu finden und die gefundenen Dokumente inklusive der Wortpositionen zu liefern. Der Typ von `search` sieht so aus:

```
1 type Document = [: String :]
2 type DocColl = [: Document :]
3 search :: [: Document :] -> String -> [: (Document, [:Int:]) :]
```

Wir beginnen mit einem leichteren Problem: dem Auffinden aller Vorkommen eines Wortes in einem einzelnen Dokument:

```
1 wordOccs :: Document -> String -> [:Int:]
2 wordOccs d s = [: i | (i,s2) <- zipP [:1..lengthP d:] d
3                     , s == s2 :]
```

Hier wird ein Filter in der Array Comprehension benutzt, der nur die Paare `(i,s2)` auswählt, für die `s == s2` gilt. Dieser Filter wird parallel angewendet, weil es sich hier um eine Array Comprehension handelt. Die Paare `(i,s2)` werden aus einem Array von Paaren ausgewählt, das durch ein `zipP` aus Dokument und einem Array aus Indizes

konstruiert wird. Es fällt auf, dass man bei DPH Parallel Arrays und Listen von der Notation her so ähnlich wie nur möglich machen will. Die Suche selbst stellt nun kein Problem mehr dar:

```
search :: [: Document :] -> String -> [: (Document, [:Int:]) :]
search ds s = [: (d,is) | d <- ds
                , let is = wordOccs d s
                , not (nullP is) :]
```

4.3 Vektorisierung

DPH basiert auf einer Transformation von Programmen verschachtelter Datenparallelität in Programme flacher Datenparallelität. Diese Transformation nennt man Vektorisierung. Sie muss sowohl auf Datenstrukturen, als auch auf Funktionen durchgeführt werden.

4.3.1 Repräsentation von Daten

Daten sollten so dargestellt werden, dass alle Parallel Arrays nur noch primitive, flache Daten, wie `Int`, `Float` oder `Double` enthalten. Ein Parallel Array von `Float` besteht aus einem kontinuierlichen Array aus Gleitkommazahlen. Den Typ Parallel Array könnte man nun, unter Verwendung von data type families, je nach Fall wie folgt definieren:

```
data instance [: Int    :] = PI Int ByteArray
data instance [: Float  :] = PF Int ByteArray
data instance [: Double :] = PD Int ByteArray
```

Das `Int` Feld repräsentiert die Größe des Arrays. Die `data` Deklarationen sind unüblicherweise nicht parametrisiert, weil die Repräsentation des Arrays abhängig ist, von den enthaltenen Elementen.

Ein Parallel Array von Paaren sollte nicht als Array von Zeigern auf Paare, die zufällig im Hauptspeicher verteilt sind, repräsentiert werden. Man erhält bessere Lokalität, wenn eine Repräsentation durch ein Paar von Arrays erfolgt:

```
data instance [: (a,b) :] = PP [:a:] [:b:]
```

Bei einem Parallel Array von Parallel Arrays muss wieder auf ein Array von Zeigern verzichtet werden. Stattdessen werden die enthaltenen Arrays konkateniert, zusammen mit einem Array, das Indizes auf die Anfänge der enthaltenen Arrays enthält.

```
data instance [: [:a:] :] = PA [:Int:] [:a:]
```

Wenn man sich jetzt eine dünnbesetzte Matrix vorstellt, die aus dünnbesetzten Vektoren besteht:

```
type SparseVector = [: (Int, Float) :]
type SparseMatrix = [: SparseVector :]
```

Eine Instanz davon könnte so aussehen:

```
m :: SparseMatrix
m = [: [:(1,2.0), (7,1.9):], [:(3,3.0:] :]
```

Die Repräsentation in DPH sieht so aus:

```
PA [:0,2] (PP [:1,   7,   3  :]
              [:2.0, 1.9, 3.0:])
```

4.3.2 Vektorisieren des Codes

Nachdem der syntaktische Zucker der Array Comprehensions durch einfachere Funktionen, wie `mapP` ersetzt wurde, müssen die vorhandenen Funktionen in eine Form gebracht werden, in der sie auf Parallel Arrays operieren können.

Betrachten wir folgende Funktion:

```
f :: Float -> Float
f x = x*x +1
```

Für jede Funktion wird eine geliftete Version erzeugt:

```
fL :: [:Float:] -> [:Float:]
fL x = (x *L x) +L (replicateP n 1)
  where
    n = lenghtP x
```

Intern nutzt `fL` Vektorinstruktionen, wie `+L`

```
+L :: [:Float:] -> [:Float:] -> [:Float:]
```

Die Konstante `1` muss ebenfalls repliziert werden, damit es als Argument von `+L` den richtigen Typ hat. Grob gesagt wird beim Lifting folgendes gemacht:

- Konstanten durch `replicateP ersetzen`
- Funktionen durch ihrer gelifteten Versionen ersetzen, z. B. `+` wird zu `+L`
- Parameter bleiben

Die neue Definition entspricht der Gleichung `fL = mapP f`, damit wird ein Array auf ein Array abgebildet. Im Endeffekt handelt es sich um eine spezialisierte Variante von `mapP` dahingehend, dass das Funktionsargument fixiert wird. Die Idee ist jetzt, dass jeder Aufruf von `mapP f` ersetzt wird durch `fL`.

Dennoch gibt es ein Problem: Bei verschachtelten Funktionen kann folgender Fall auftreten. Angenommen wir haben

```
g :: [:Float:] -> [:Float:]
g xs = mapP f xs
```

Zunächst ersetzen wir `mapP f` durch `fL`

```
g :: [:Float:] -> [:Float:]
g xs = fL xs
```

Aber jetzt müssen wir g auch liften, für den Fall, dass es Aufrufe der Form `mapP g` gibt:

```
gL :: [:[:Float:]:] -> [:[:Float:]:]
gL xs = fLL xs
```

Das Problem ist, dass wir jetzt eine doppelt geliftete Version von `f` bräuchten. Die Lösung ist eine Definition von `fLL` durch `fL`:

```
1 fLL :: [:[:Float:]:] -> [:[:Float:]:]
2 fLL xss = unconcatP xss (fL (concatP xss))
```

Die Funktion `concatP` konkateniert erst alle Reihen von `xss` zu einem einzigen flachen Array. Die Implementierung sieht so aus:

```
1 concatPA :: [: [:a:] :] -> [:a:]
2 concatPA (PA segs xs) = xs
```

D. h. durch die bereits flache Repräsentation der Daten ist hier nichts zu berechnen. Dannach wird `map f` auf diesem Array ausgefürt. Zum Schluss wird das Ergebnis mit `unconcatP` wieder in die ursprüngliche Form gebracht:

```
1 unconcatP :: [:a:] -> [:a:] -> [: [:a:] :]
2 unconcatP (PA segs xs) ys = PA segs ys
```

Dannach wird das Ergebnis mithilfe des Segment-Arrays aus der ursprünglichen Datenstruktur wieder in die ursprüngliche Form gebracht. Bemerkenswert ist, dass `xss` Arrays unterschiedlicher Länge beinhalten kann. Trotzdem haben die Operationen `concatP` und `unconcatP` lediglich konstante Laufzeit in der Implementierung von DPH.

4.4 Performanz

In [Cha+07] wurden Messungen zur Performanz von DPH veröffentlicht. Die Tests wurden an einer Implementierung eines Programms zur Multiplikation von dünnbesetzten Matrizen mit Vektoren durchgeführt (Abbildung 4.1). Auch wenn das Programm möglicherweise nicht der Realität entspricht, so ist der lineare Speedup doch bemerkenswert.

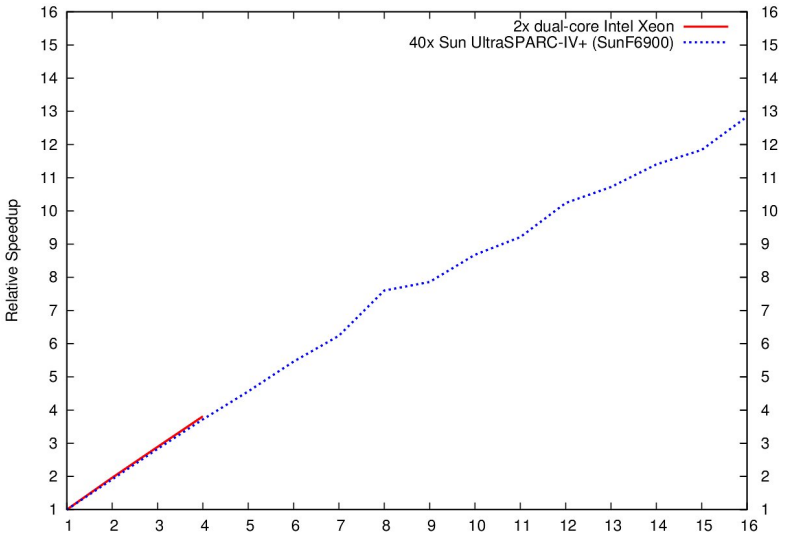

Abbildung 4.1: Performanz in DPH. [Cha+07]

Literatur

[Ble96] G.E. Blelloch. "Programming parallel algorithms". In: *Communications of the ACM* 39.3 (1996), S. 85–97. ISSN: 0001-0782.

[Cha+07] M.M.T. Chakravarty u. a. "Data Parallel Haskell: a status report". In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. ACM. 2007, S. 10–18.

[Dis+06] A. Discolo u. a. "Lock free data structures using STM in Haskell". In: *Functional and Logic Programming* (2006), S. 65–80.

[Har+05] T. Harris u. a. "Composable memory transactions". In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2005, S. 48–60. ISBN: 1595930809.

[Mar+10] S. Marlow u. a. "Seq no more: better strategies for parallel Haskell". In: *Proceedings of the third ACM Haskell symposium on Haskell*. ACM. 2010, S. 91–102.

[OSG11] B. O'Sullivan, D. Stewart und J. Goerzen. *Real World Haskell*. Jan. 2011. URL: `http://book.realworldhaskell.org/read/`.

[PJ+08] S. Peyton Jones u. a. "Harnessing the Multicores: Nested Data Parallelism in Haskell". In: *FSTTCS*. 2008, S. 383–414.

[PJGF96] S. Peyton Jones, A. Gordon und S. Finne. "Concurrent Haskell". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1996, S. 295–308. ISBN: 0897917693.

[PJS09] S. Peyton Jones und S. Singh. "A tutorial on parallel and concurrent programming in haskell". In: *Advanced Functional Programming* (2009), S. 267–305.

[Wika] *Data parallelism*. Jan. 2011. URL: `http://de.wikipedia.org/wiki/Data_parallelism`.

[Wikb] *Explicit Parallelism*. Jan. 2011. URL: `http://en.wikipedia.org/wiki/Explicit_parallelism`.

[Wikc] *Implicit Parallelism*. Jan. 2011. URL: `http://en.wikipedia.org/wiki/Implicit_parallelism`.