Seminarausarbeitung AW2

Kjell Otto

Concurrent Programming Models
Related Work

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einführung 1

2 Nebenläufige Programmiermodelle 2

2.1 Parallele Hardwarearchitekturen . . . 2

2.2 Related Work . . . 3

2.2.1 Aktoren in Scala . . . 3

2.2.2 Software Transactional Memory mit C# . . . 5

2.2.3 MapReduce von Google . . . 7

3 Fazit 11

3.1 Zusammenfassung . . . 11

3.2 Ausblick . . . 11

Literaturverzeichnis 12

1 Einführung

Der Begriff des „ubiquitous computing“ von Mark Weiser beschreibt einen natürlichen Umgang mit Computer, wie er heute schon weit fortgeschritten ist. Die Bedienung und Nutzung von Computern fließt in das tägliche Leben ein und die auf ihnen basierenden Dienste werden immer selbstverständlicher, vgl. [14]. Der Fortschritt von kleiner und effizienter werdender Hardware wird weiter verfolgt und die Softwareentwicklung muss sich diesem Trend anpassen. Je mehr Computer an einem Dienst mitarbeiten desto mehr Synchronisation muss betrieben werden um einen sicheren Zugriff auf geteilte Ressourcen zu gewährleisten. Der steigende Aufwand in Synchronisationsmechanismen birgt Gefahren von undurchsichtigen Systemzuständen bis hin zu nicht reproduzierbaren Szenarien. Mit der Komplexität der Hardware steigt die Nachfrage der Softwareentwickler nach Mechanismen zur Automatisierung oder Vereinfachung von komplexen Synchronisationenszenarien.

Die in [8] vorgestellten Konzepte aus der Programmiersprache Clojure wurden zur Lösung von Problemen in der Entwicklung nebenläufigen Software herangezogen. Diese Mechanismen wurden im Detail erläutert und deren Vor- und Nachteile hervorgehoben. In dieser Seminararbeit sollen andere Mechanismen und Konzepte vorgestellt werden um Probleme bei der effizienten nebenäufigen Datenverarbeitung zu lösen. Dabei wurden Projekte aus verschiedenen Bereichen herangezogen um einen Überblick über die aktuellen Technologien zu geben.

Im Living Place Hamburg, einem Forschungsprojekt zum Thema zukünftiges Leben, wird die in [14] beschriebene Situation des allgegewärtigen Computers schon Heute entstehen und es bietet damit eine Forschungsgrundlage zur Erprobung von Synchonisationsmechanismen in einem hochparallelen Umfeld, vgl. [12]. Die in diesem Projekt entstandene Architektur wird mit einer Vielzahl von Prozessoren unterstützt die Instrumentalisiert werden sollen. Für diesen Einsatz werden Hilfestellungen der aktuellen Forschung zum Thema Synchonisationsmechanismen untersucht und analysiert. Ein Schwerpunkt bei den Untersuchungen soll die Verständlichkeit und Fehleranfälligkeit der Konzepte im alltäglichen Einsatz sein.

Im 2. Kapitel werden die Programmiermodelle und Hardwarearchitekturen die der aktuellen Softwareentwicklung zur Verfügung stehen erläutert und unterteilt. Aus diesen Kategorien wurden drei Projekte ausgewählt um beispielhaft, im Abschnitt 2.2, Konzepte aus den Bereichen zu erläutern und jeweilige Vor- und Nachteile hervorzuheben. Den Abschluss dieser Arbeit bildet eine Zusammenfassung mit Ausblick auf den Einsatz der erläuterten Technologien im Forschungsprojekt Living Place Hamburg.

2 Nebenläufige Programmiermodelle

Die Programmiermodelle für nebenläufige Applikationen sind grundsätzlich in zwei Kategorien von Hardwarearchitekturen zu unterteilen. Zum einen die Multicore-Mechanismen und zum anderen die Multinode-Mechanismen. Dabei handelt es sich bei Multicore-Mechanismen um Programmiermodelle die mehrere CPUs auf einem Computer instrumentalisieren und bei Multinode-Mechanismen um Programmiermodelle die mehrere Computer instrumentalisieren, vgl. [1]. Der folgende Abschnitt wird diese zwei Kategorien beschreiben und drei Programmiermodelle vorstellen die sich diesen Kategorien unterordnen.

2.1 Parallele Hardwarearchitekturen

Allgemein lassen sich Softwarearchitekturen nach ihrer Anzahl von parallelen Kontroll- und Datenflüssen in eine der folgenden Kategorien unterteilen:

- *Single Instruction Single Data (SISD)* Sequenzielle Verarbeitung von Instruktionen auf immer unterschiedlichen Datenflüssen, sodass keine Parallelisierung möglich ist(z.B. 1-Core PC)
- *Single Instruction, Multiple Data (SIMD)* Parallele Verarbeitung von einer Instruktion auf mehreren unterschiedlichen Datenflüssen(z.B. Grafikkarten)
- *Multiple Instruction, Single Data (MISD)* Verschiedene Instruktionen auf einem Datum, ein eher theoretisches Modell, was sich für die Parallelisierung nicht eignet(ein eher theoretisches Modell zu dem keine Hardware bekannt ist)
- *Multiple Instruction, Multiple Data (MIMD)* Systeme die unabhängig voneinander auf unterschiedlichen Datenflüssen arbeiten. Diese Kategorie beschreibt die o.g. zwei Architekturen die heute am weitesten verbreitet sind. Es wird zwischen verteiltem(z.B. Cluster) und gemeinsamem(z.B. Multicores) Speicher unterschieden.

Diese Architekturen können auch kombiniert werden, sodass z.B. eine MIMD-Architektur die Instruktionsverteilung auf mehrere Computer darstellt und auf den verteilten Computern SIMD-Konzepte zum Einsatz kommen, vgl. [6].

Hardwarehersteller werden auch in Zukunft weiter auf die Entwicklung von Multicores eingehen, sodass sich die Anzahl der Kerne innerhalb der CPUs erhöhen wird. Der Softwareentwicklung stehen heute schon in der Standardhardware für Heimcomputer mehrere Kerne zur Verfügung und eine Aufgabe wird deren Instrumentalisierung sein, vgl. [8].

2.2 Related Work

Die folgenden drei Projekte bzw. Programmiermodelle geben Einblick in Konzepte zur Instrumentalisierung von mehreren Kernen in einem oder mehreren Computern. Dabei sind die ersten beiden Arbeiten darauf ausgerichtet auf einem Computer alle Kerne zur Berechnung zu verwenden und die letzte Arbeit zielt auf die Verwendung von Computernetzen(Clustern) ab.

2.2.1 Aktoren in Scala

Scala ist eine Hybridsprache die objektorientiert und funktional zu gleich ist. Sie wird seit 2001 entwickelt und ist statisch typisiert. Scala läuft auf der Java Virtual Machine (JVM) und ist voll kompatibel zu allen bestehenden Java Bibliotheken. Der Synchronisierungsmechanismus für parallele Berechnungen in Scala ist, neben den aus Java verfügbaren Mechanismen, der Aktor , vgl. [10].

Aktoren

Das Aktormodell ist seit 1973 ein Modell zur Vereinfachung von parallelem Rechnen, vgl. [5]. Das Modell setzt zur Vereinfachung einen einzigen Mechanismus ein, den Aktor. Der Aktor ist ein Konstrukt was ähnlich wie ein Thread, als berechnende Einheit fungiert. Aktoren dienen als unabhängige funktionsausführende Abschnitte die auf Nachrichten reagieren. Jeder Aktor empfängt Nachrichten über seine Message-Queue und sendet Nachrichten an Message-Queues anderer Aktoren, vgl. Abb. 2.1. Sie können parallel folgende Funktionen, als Reaktion auf eine Nachricht ausführen:

- Nachrichten versenden
- neue Aktoren erstellen
- ein Verhalten für den Empfang der nächsten Nachricht festlegen

Es gibt keine vorgeschriebene Reihenfolge in der Verarbeitung dieser Möglichkeiten und sie können parallel ausgeführt werden. Indem jeder Aktor in einem von der Außenwelt unbeeinflussten Speicher arbeitet, benötigt dieser Mechanismus keine Synchronisationsmechanismen wie man sie von z.B. C++ oder Java kennt, vgl. [8].

Scalas Aktoren

In Scala ist das Aktormodell kein Sprachbestandteil wie etwa bei Erlang, sondern Teil der Standardbibliothek seit Version 2.1.7. Die Implementierung orientiert sich zwar an der von Erlang und übernimmt die wesentlichen Charakteristika und Operationen, unterscheidet sich aber in einem wesentlichen Punkt. In Erlang bietet die Virtuelle Maschine sog. *lightweight-threads* an, die nicht auf Betriebsystemthreads übersetzt werden, sondern leichtgewichtigere VM-Threads darstellen. In Scala werden die Aktoren auf JVM-Threads

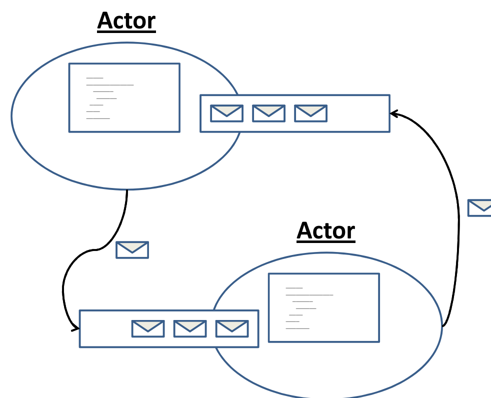

Abb. 2.1: Symbolische Darstellung von zwei Aktoren die sich Nachrichten an ihre Message-Queues zusenden

übertragen, welche letztendlich auf native Betriebssystemthreads zurückgreifen. Der Pool von Scala-Workerthreads ist zunächst mit 4 Threads ausgestattet und wird dynamisch erweitert sobald es mehr aktive Aktoren gibt als zur Verfügung stehende Threads. Die meisten Implementationen von JVMs können nicht mehr als tausende von Threads händeln, vgl. [9]. Scala bietet eine Möglichkeit um dieses begrenzende Problem zu umgehen, indem man einen eventbasierten Ansatz verfolgt. Im Gegensatz zu Erlang kann die Aktorbibliothek in Scala allerdings nicht sicherstellen, dass Aktoren voneinander isoliert sind und ausschließlich durch asynchrone Nachrichten kommunizieren, vgl. [2]. Um das Problem der eventuellen Manipulation von Daten ohne den Aktor zu beheben, bietet Scala zwei Arten von Datenstrukturen an, *var* und *val*. Eine *var* in Scala ist eine Variable wie sie aus Java bekannt ist, sie kann verändert und überschrieben werden. Die *val* Datenstruktur ähnelt allerdings eher einer Konstanten(oder in Java *final*), sie kann, einmal gesetzt, nicht mehr verändert werden(vgl. Immutable [8]) und ist mit dieser Eigenschaft besonders für den Einsatz im seiteneffektfreien Arbeiten mit Aktoren geeignet.

Aktoren werden in Scala definiert indem man von der Klasse *scala.actors.Actor* ableitet und die *act()* Methode überschreibt:

```
0 class TestActor(number: Int, test: Actor) extends Actor {
    def act() {
      while (true) {
        receive {
          ...
5       }
      }
    }
  }
```

Die eventbasierte Methode zeigt das folgende Listing:

```
0 class TestActor(number: Int, test: Actor) extends Actor {
    def act() {
      loop {
        react {
          ...
5       }
      }
    }
  }
```

Zur netzwerkübergreifenden Kommunikation von Aktoren bietet Scala die Bibliothek *scala.actors.remote*.

Bewertung

Scala ist eine, im Vergleich zu Java, recht junge Sprache und wird ständig weiterentwickelt. Zur Realisierung von hochparallelen Applikationen bietet sie die Java-Mechanismen an und erweitert diese mit dem Aktorkonzept aus der eigenen Standardbibliothek. Aktoren in Scala bieten einen einfachen Mechanismus um spezifische Programmabschnitte zur Parallelität zu führen.

Im Projekt kann Scala gut eingesetzt werden, da die nahtlose Interoperabilität mit Java den Zugang zu nahezu allen Projektkomponenten bietet. Problematisch in Scala ist jedoch, dass nicht alle Datenstrukturen unveränderlich sind und dem eventuellen Neueinsteiger in ein Projekt die selben Synchronisationsfehler ermöglichen wie sie auch in Java möglich sind.

2.2.2 Software Transactional Memory mit C#

C# ist eine Programmiersprache der .Net Familie und läuft in der Common Language Runtime (CLR) von Microsoft. Als objektorientierte Sprache unterstützt C# die Konzepte der Kapselung, Vererbung und Polymorphie. Außerdem kennt C# auch Delegates und Covariance im Typsystem. Zur Entwicklung paralleler Applikationen bietet die Sprache C# Mutex, Semaphor und andere aus Java bekannte Mechanismen zu Synchronisation von geteilten Ressourcen. Zusätzlich bietet das .Net Framework allerdings seit dem 28 Juli 2009 die sog. STM.Net für C# an. Die STM.Net gibt dem C# Entwickler ein neues Syntaxelement, den *Atomic.Do(() => {...}* Codeblock. Dieser Codeblock wird benutzt um bestimmte Bereiche des Programms als atomar zu kennzeichnen. Dabei sorgt nicht der Entwickler, sondern die STM.Net dafür, dass diese Bereiche atomar ausgeführt werden. Im Folgenden werden beispielhaft Eigenschaften von Software Transactional Memory Systemen und die konkrete Implementation eines STMs im .Net Framework genauer erläutert.

Software Transactional Memory

Der Software Transactional Memory (STM) ist ein Mechanismus zur Kontrolle von parallelen Zugriffen auf geteilten Speicher, ähnlich den Transaktionen auf Datenbanken. Der .Net STM dient hier als Beispielimplementation einer STM, wobei die Implementierungen von STMs Gegenstand aktueller Forschung sind, vgl. [13]. Eine Transaktion ist in diesem Kontext als Folge von Befehlen und Programmteilen die lesend und/oder schreibend auf geteilte Daten zugreifen. Diese Transaktionen vereinen die aus den Datenbanken bekannten Eigenschaften von ACID(Atomic, Consistent, Isolated und Durable). Die Folgen von Transaktionen werden für den außerhalb der Transaktion liegenden Programmteil zu einem bestimmten Zeitpunkt auf einmal, von der Außenwelt unbeinflusst und konsistent sichtbar. Sollte eine Transaktion keinen Erfolg erzielen wird keine der Änderungen die sie zur Folge

gehabt hätte sichtbar, sodass selbst wenn Teile der Transaktion erfolgreich waren, diese Folgen zurückgesetzt werden. Um einen solchen Mechanismus anzubieten bedienen sich die STMs indirekter Referenzierung, vgl. Abb. 2.2. Durch die indirekten Referenzen kann der STM die Objekte schützen, indem er den Zugriff auf Ressourcen oder Objekte durch definierte Zugriffsmechanismen beschränkt.

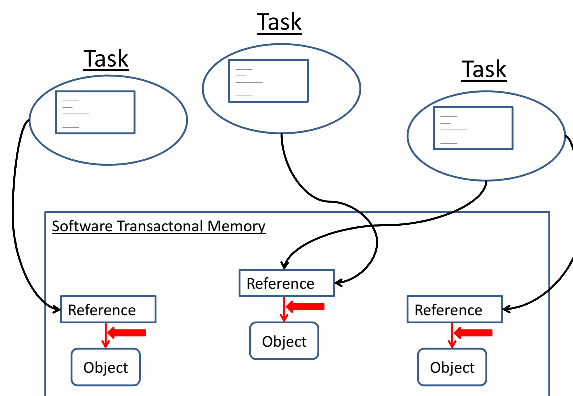

Abb. 2.2: Symbolische Darstellung eines Software Transactional Memory der die Zugriffe auf Objekte über indirekte Referenzen schützt

STM.Net

Der Software Transactional Memory vom .Net Framework hat einige Änderungen an der CLR und dessen Just In Time(JIT)-Compiler mit sich gebracht. Die CLR wurde so angepasst, dass es nicht nötig ist Objekte die mit dem STM geschützt werden sollen, mit bestimmten Typen zu kennzeichnen, wie zum Beispiel *TransactionalInt*. Die .Net Implementation ist in der Lage jeden einfachen oder komplexen Typen, durch die automatische Erweiterung des Scopes in einem Transaktionsblock, zu schützen. Dabei kennzeichnet der Entwickler einen Codeblock mit dem *Atomic.Do()*-Aufruf und der CLR-JIT kann dynamisch alle Objekte die in diesem Abschnitt bearbeitet werden erkennen. Für diese Objekte werden dann automatisch Schattenkopien und feingranulare Locks erstellt um die Eigenschaften von ACID zu gewährleisten. Durch diese automatischen Mechanismen wurde die Bytecodeinterpretation des CLRs verändert und wenn man diese einsetzen will, benötigt man die angepasste CLR, vgl. [7].

Bewertung

Die STM.Net Implementation eines Software Transactional Memorys bietet eine einfache Möglichkeit zur Synchronisation von geteilten Ressourcen. Im Projekt Living Place werden die meisten Komponenten als Basis die JVM nutzen und es würde zur Interaktion mit Programmen auf Basis des .Net-Frameworks nur die Möglichkeit des Message-Passing geben, vgl. [12]. Des Weiteren wurde die angepasste CLR zur Realisierung des STM nicht

in die aktuelle Produktreihe(VS2010) übernommen und ist damit nicht für die Softwareentwicklung zu empfehlen, vgl. [4].

2.2.3 MapReduce von Google

Das MapReduce Projekt von Google wurde entwickelt um große Datenmengen, mehrere Terrabyte, zu verarbeiten und überschüssige Rechenkapazitäten in großen Rechenzentren effektiver zu nutzen. Die Idee hinter dem Konzept ist die Aufteilung der Verarbeitungsschritte in zwei Funktionen, die Map-Funktion und die Reduce-Funktion. Ursprünglich kommen diese Funktionen aus dem Lisp-Umfeld und bieten eine generelle Abstraktion vom Anwenden einer Funktion auf eine Sammlung von Daten. Die Map-Funktion ist eine Funktion die auf alle Teile einer Sammlung angewendet, *gemapped* wird. Die Reduce-Funktion ist eine Funktion mit deren Anwendung die Sammlung auf ein einzelnes Ergebnis *reduziert* wird. Google wendet dieses Verarbeitungsprinzip heute in allen Rechenzentren an und vereinfacht die Entwicklung von Programmen die große Datenmengen verarbeiten für ihre Entwickler, vgl. [3].

Ein sog. Task, eine Aufgabe die das Programm erfüllen soll, wird zunächst in die Map- und Reduce-Funktion unterteilt. Eine Node(ein Rechner im Cluster) übernimmt die Aufgabe des koordinierenden Masters. Dieser verteilt die Funktionen auf Nodes mit freien Ressourcen und weist diesen Teilaufgaben zu, vgl. Abb. 2.3.

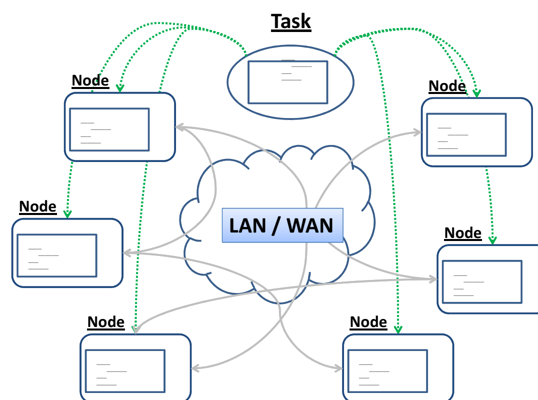

Abb. 2.3: Symbolische Darstellung der Verteilung eines Tasks auf mehrere Nodes zur gemeinsamen Berechnung eines Ergebnisses

MapReduce

Das MapReduce-Konzept basiert auf den zwei Funktionen „Map“ und „Reduce“. Beide Funktionen müssen seiteneffektfrei und auf alle Daten die verarbeitet werden sollen anwendbar sein. Außerdem sind diese Funktionen vom Entwickler zu implementieren und aufgabenspezifisch anzupassen. Die Eingabedaten für die Map-Funktion sind

Schlüssel/Werte-Paare die nach Anwendung der Funktion wieder auf Schlüssel/Werte-Paare abgebildet werden. Dieses Ergebnis dient als Zwischenergebnis und Eingabewert für die Reduce-Funktion. Die Zwischenergebnisse werden mittels Iteratoren an die Reduce-Funktion übergeben, um Teilbereiche bei eventuell zu großen Ergebnismengen nicht im Speicher halten zu müssen. Die Reduce-Funktion errechnet aus dem Zwischenergebnis ein Endergebnis. Diese Liste wird im nächsten Schritt zu einem Ergebnis pro Eingabe-Schlüsselwert zusammengefasst. Es entsteht eine Ergebnisliste die für jeden Schlüsselwert ein Ergebnis liefert. Das Ergebnis ist, soweit nicht durch besondere Parameterisierung verhindert, auf alle Reduce-Funktionen berechnenden Computer verteilt.

In diesem Framework kann der Benutzer parametrisieren wie viele beteiligte Nodes es gibt, in wie viele Teile die Aufgabe zerlegt wird und wie viele Teilergebnisse entsprechend produziert werden. Unter Anderem kann spezifiziert werden das für den Task M Map-splits und R Reduce-splits zum Einsatz kommen sollen. Dabei sind typische Werte $M = 200,000$ und $R = 5,000$ bei $2,000$ Nodes. Die Verarbeitung von splits wird in folgenden Schritten auf alle beteiligten Nodes verteilt, vgl. Abb. 2.4:

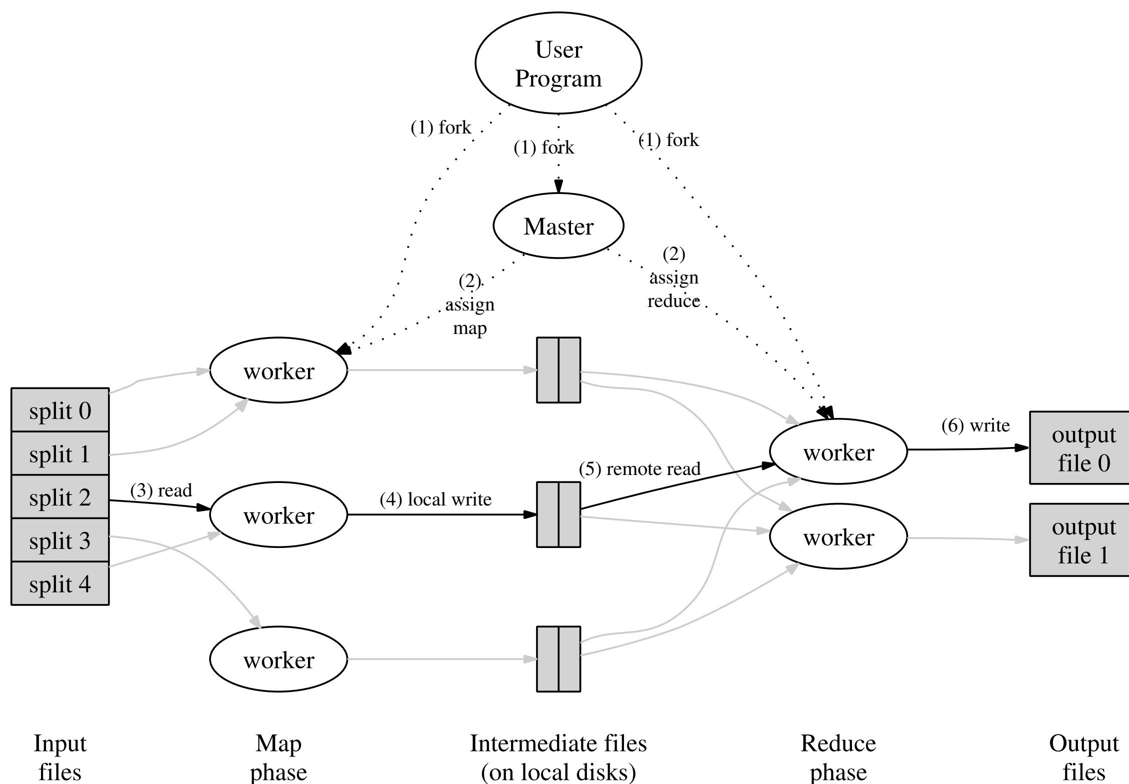

Abb. 2.4: Schematische Darstellung der Ausführung eines MapReduce-Tasks [3]

1. Ausgehend vom Programm des Entwicklers, welches die Map- und Reduce-Funktion definiert, teilt die Bibliothek die Eingabedaten in typischerweise 16 bis 64 MB große Pakete und startet das Programm auf allen verfügbaren Rechnern im Cluster.

2. Eines der gestarteten Programme ist der Master. Der Master übernimmt die Verwaltung, also Verteilung von Teilaufgaben an die sog. „worker“, die anderen Kopien des

Programms. Der Master wählt untätige worker aus und verteilt Map- oder Reduce-Aufgaben an diese.

3. Ein worker der einen Map-Task zugewiesen bekommt, liest den dazugehörigen Eingabesplit und analysiert die Schlüssel/Werte-Paare aus dem split um sie an die entwicklerdefinierte Map-Funktion zu übergeben. Diese Zwischenergebnisse der Map-Funktion werden im Speicher gepuffert.

4. Die im Speicher gehaltenen Daten werden periodisch, unter Anwendung der Partitionierungsfunktion, auf die Festplatte geschrieben. Die Adressen der Daten auf der lokalen Festplatte werden an den Master zur weiteren Verwendung übertragen. Der Master ist dann in der Lage, diese Adressen an worker die Reduce-Tasks zugewiesen bekommen haben, zu übergeben.

5. Wenn ein worker vom Master den Reduce-Task mit den zugehörigen Adressen, den Ergebnissen der Map-worker, zugewiesen bekommt, beginnt er mittels Remote Procedure Calls (RPC) von den Map-workern die Zwischenergebnisse zu lesen. Wenn der Reduce-worker alle Zwischenergebnisse gelesen hat beginnt er mit dem Sortieren um alle Schlüsselwerte zu gruppieren. Dadurch kann bei Ausführung der Reduce-Funktion davon ausgegangen werden, dass gleiche Schlüsselwerte aufeinander folgen und miteinander reduziert werden können. Sollte das Sortieren zu viel Speicher beanspruchen wird auf ein Sortierverfahren ausgewichen das im Festspeicher sortiert.

6. Der Reduce-worker iteriert über die sortierten Zwischenergebnisse und übergibt zu jedem einzelnen Schlüssel alle Werte an die Reduce-Funktion des Entwicklers. Das Ergebnis ist die Liste mit jeweils einem Ergebnis pro Schlüssel. Diese Ergebnisliste wird an die Datei die das Endergebnis darstellt angehängt.

7. Wenn alle Map- und Reduce-Tasks abgeschlossen sind kehrt das Programm zurück zum Entwicklercode. Bei erfolgreichem Abschließen der Gesamtaufgabe sind die Ergebnisse der Reduce-Funktionen als Dateien bei den Reduce-workern hinterlegt.

Normalerweise werden die Ergebnisse nicht zusammengetragen, denn die Folgeaufgaben werden meist auch mit dem MapReduce-Framework, welches in der Lage ist mit unzusammenhängenden Ergebnissen umzugehen, berechnet. Sollten die Ergebnisse zusammengetragen werden müssen, so könnte man R für diesen MapReduce-Task mit 1 parameterisieren, vgl. [3].

Bewertung

Das MapReduce-Framework von Google ist geeignet um viele Computer mit ungenutzten Rechenkapazitäten zu instrumentalisieren. Es wird erfolgreich von Google verwendet um bis zu 2000 Computer auf einmal zur Verarbeitung von großen Datenmengen zu nutzen. Die Aufgaben die mit dem Framework berechnet werden können müssen allerdings in das MapReduce-Konzept passen. Das von Google beschriebene Framework wurde auch auf andere Plattformen übertragen und es gibt erfolgreiche Implementationen z.B. mit Hadoop auf der Java-Plattform, vgl. [11]. Sobald die Datenmengen im Living Place steigen können

mit dem MapReduce Konzept auch große Datenmengen analysiert und verarbeitet werden. Die Voraussetzungen dazu sind durch die Architektur der Datenhaltung gegeben und ermöglichen den Einsatz von vielen im Labor vorhandenen Computern, vgl. [12].

3 Fazit

3.1 Zusammenfassung

Die hier vorgestellten Arbeiten zeigen drei unterschiedliche Mechanismen um mit der steigenden Parallelität in Computern umzugehen. Die Ziele der Projekte variieren dabei von der Instrumentalisierung von CPUs durch die Programmstruktur über automatische Referenzmechanismen bis hin zu einer Verteilung der Aufgabe über ein ganzes Rechenzentrum. Die Scala Aktoren bieten einen Einblick in ein Konzept das schon lange im Einsatz ist und mit dem viele hoch parallele Programme implementiert wurden, wie z.B. der RabbitMQ Message-Broker, vgl. [12]. Die STM.Net dagegen bietet dem Entwickler die Möglichkeit die Verantwortung über die Referenzverwaltung nahezu ganz abzugeben und erleichtert das Erstellen von parallelen Programmen durch seine Automatismen. MapReduce von Google versucht dagegen viele Computer zu nutzen um eine Aufgabe effizient zu lösen. Keiner der Ansätze bietet eine Lösung der Probleme die auftreten wenn man Programme parallelisieren möchte, aber die gezeigten Teillösungen bieten Anlass zu weiteren Untersuchungen auf dem Gebiet.

3.2 Ausblick

Der Living Place Hamburg als Forschungslabor für zukünftiges Leben bietet ein Umfeld in dem eine Vielzahl an Sensoren zum Einsatz kommen wird. Alle Sensoren werden Daten liefern die eingesetzt werden um das Verhalten der Bewohner zu studieren oder diese in ihrem täglichen Leben zu unterstützen. Die dabei aufgenommenen Daten werden durch viele Programme verarbeitet und genutzt um Entscheidungen zu fällen oder z.B. die Beleuchtung zu regulieren. Diese Verarbeitung bietet ein sehr rechenintensives und eventuell auch zeitkritisches Umfeld in dem Parallelisierungsmechanismen auch in Zukunft von Nöten sein werden um die Berechnung aller Aufgaben zu gewährleisten. Des Weiteren möchte ich weiterhin am Kommunikationsmittelpunkt des Forschungslabors, der Kommunikationsschnittstelle, arbeiten und die effiziente Verarbeitung der Daten auch in höheren Abstraktionsschichten gewährleisten.

Literaturverzeichnis

[1] BAUKE, Heiko ; MERTENS, Stephan: *Cluster Computing*. 2006. – ISBN 3-540-42299-4

[2] CHAROUSSET, Dominik: Softwarearchitekturen für die Entwicklung verteilter Anwendungen. In: *Ausarbeitung Anwendungen 2 SoSe 2009* (2009)

[3] DEAN, Jeffrey ; GHEMAWAT, Sanjay: *MapReduce: Simplifed Data Processing on Large Clusters*. Google, Labs. 2010. – URL http://labs.google.com/papers/mapreduce.html. – [Online; Stand 23. Mai 2010]

[4] GROFF, Dana: *STM.NET DevLab Incubation Complete*. 2010. – URL http://blogs.msdn.com/b/stmteam/archive/2010/05/12/stm-net-devlab-incubation-complete.aspx. – [Online; Stand 28. August 2010]

[5] HEWITT, Carl ; BISHOP, Peter ; STEIGER, Richard: A universal modular ACTOR formalism for artificial intelligence. In: *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1973, S. 235–245

[6] HOFFMANN S. ; LIENHART, R.: *OpenMP*. 2008. – ISBN 978-3-540-73122-1

[7] NEWARD, Ted: *ACID Transactions with STM.NET*. 2010. – URL http://msdn.microsoft.com/en-us/magazine/ee291549.aspx. – [Online; Stand 28. August 2010]

[8] OTTO, Kjell: Clojure - concurrency revisited. In: *Hochschule für Angewandte Wissenschaften Hamburg* (2010). – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master09-10-aw1/otto/bericht.pdf. – [Online; Stand 05. Februar 2010]

[9] SCALA-LANG.ORG: *Scala Actors: A Short Tutorial*. 2010. – URL http://www.scala-lang.org/node/242. – [Online; Stand 19. Mai 2010]

[10] SCALA-LANG.ORG: *A Tour of Scala*. 2010. – URL http://www.scala-lang.org/node/104. – [Online; Stand 19. Mai 2010]

[11] THE APACHE SOFTWARE FOUNDATION: *MapReduce: Simplifed Data Processing on Large Clusters*. The Apache Software Foundation. 2010. – URL http://hadoop.apache.org/. – [Online; Stand 28. August 2010]

[12] VOSKUHL, Sören ; OTTO, Kjell: Entwicklung einer Architektur für den Living Place Hamburg. In: *Hochschule für Angewandte Wissenschaften Hamburg* (2010). – URL `http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2010-proj/voskuhl-otto.pdf`. – [Online; Stand 31. August 2010]

[13] WANG, Xiaoqun ; JI, Zhenzhou ; FU, Chen ; HU, Mingzeng ; YANG, Xiaozong: Software Transactional Memory in Multicore Processors, dec. 2009, S. 1 –4

[14] WEISER, Mark ; BROWN, John S.: *THE COMING AGE OF CALM TECHNOLOGY*. 1996. – URL `http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm`. – [Online; Stand 31. August 2010]