X10 Language

Mihail Dzhurev

30. July 2010

Zusammenfassung

In den letzten Jahren ist die Notwendigkeit zur parallelen Verarbeitung deutlich geworden. Da die CPU-Entwicklung an die Grenzen der Betriebsfrequenz gestoßen ist, ist die Erhöhung der Anzahl der Prozessoren der einzige Weg, um deutliche Verbesserung zu herbei zu führen. Dadurch erhöht sich aber die Last auf die Programmierer, weil das Schreiben von parallelem Code deutlich schwieriger ist, als im sequentiellen Fall. Die üblichen Werkzeuge zwingen den Programmierer dazu sich mit Low-Level-Problemen wie Cache-Optimierungen oder manuellem Message Passing zu beschäftigen. Hierbei gibt es zwei grundsätzliche Probleme. Zum einen sind die Grundlagen sehr komplex, was wiederum zu einer niedrigen Produktivität der Programierer führt. Einige dieser Probleme lassen sich zumindest teilweise mit der Einführung einer neuen Sprache, die im Grunde parallel ist, lösen. Gemeinsam mit anderen Sprachen, die von der DARPA High Productivity Computing Systems (HPCS) Programms finanziert wird, ist X10 IBM's Versuch, eine Hochsprache zu entwickeln, die auch parallele Programmierung ermöglicht.

1. Einleitung

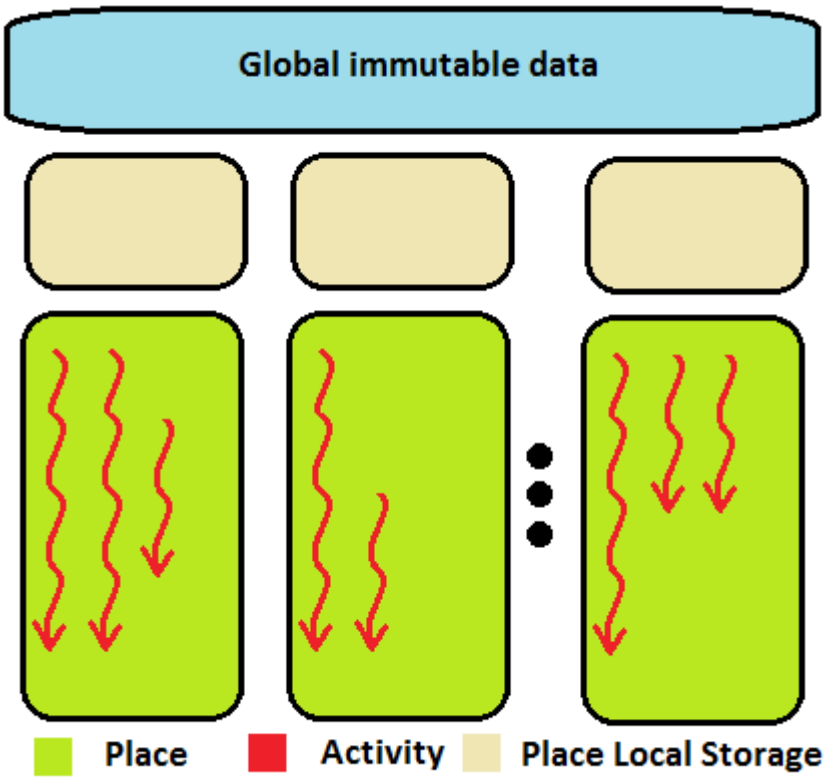
Die X10 Sprache, nicht mit dem X10 Standart zu verwechseln, wird seit 2004 bei IBM als eine Objekt orientierte Hochsprache entwickelt. Die ersten Versionen basieren auf Java, weshalb die Objekt orientierten Eigenschaften sehr ähnlich sind. Spätere Versionen haben auch eine gewisse Ähnlichkeit zu Scala, besonders die Unterstützung für funktionale Programmierung und anonyme Funktionen. Das Modell der Parallelität basiert auf dem *Partitioned Global Address Space* (PGAS). Dieses Modell versucht die Entwicklung von parallel arbeitenden Programmen zu vereinfachen. Obwohl Java die Basis für die Sprache ist, ist X10 nicht Java. Die aktuelle Version kann den X10 Programmcode in Java oder C++ mit MPI übersetzt werden. Danach kann man den übersetzten Code interpretieren oder kompilieren. Eine weitere Möglichkeit ist die Übersetzung in CUDA C, aber diese Funktion befindet sich noch im Alpha-Stadium.

2. Objekt orientierte Eigenschaften

Ähnlich zu Java, sind in X10 alle Klassen von den Wurzelklasse *Object* abgeleitet. Eine weitere Ähnlichkeit ist die automatische Speicherbereinigung. Desweiteren stehen dem Programierer Klassen, Schnittstellen (Interfaces) und Strukturen zur Verfügung. Schnittstellen spezifizieren eine Menge von abstrakten Methoden, die ein Klasse implementieren soll. X10 unterstützt keine Mehrfachvererbung von Klassen, aber ein Klasse kann mehrere Schnittstellen vererben. Strukturen sind nicht so flexibel wie Objekte, weil sie beschränkt und unveränderlich sind. Das führt dazu, dass Strukturen effektiver verwendet werden können. Es gibt keine Basistypen wie in Java, aber es gibt Strukturen für Int, Boolean, Double usw.. X10 benutzt verstärkt Container, hierbei ist es möglich die Daten von einen Container auf verschiedenen Rechnerknoten zu verteilen. X10 ist eine Sprache mit starker Typisierung, aber dem Compiler ist es möglich den Typ einer Variablen automatisch abzuleiten.

3. Das PGAS Model

Ein zentraler Begriff in diesem Modell ist der *Ort* (Place). Der globale Speicher ist in Places aufgeteilt, und jeder Place ist eine Einheit kohärenter Daten mit einer zugeordneten Anzahl an Threads. Das Modell geht davon aus, dass der Speicherzugriff in demselben Place wesentlich schneller ist als zwischen verschiedenen Places. Die Berechnung an einem Place wird durch leichte Threads durchgeführt. Diese Threads werden in X10 als *Aktivitäten* (von activity) bezeichnet. Die Aktivitäten sind so implementiert, dass sie wenig Overhead haben, so dass Hunderte von Aktivitäten gleichzeitig laufen können und auch schnell gestartet werden können. Viele Aktivitäten können innerhalb des gleichen Place gleichzeitig arbeiten, und sie haben eines festes Place. Das heißt, dass die Aktivitäten können seinen Place nicht während der Laufzeit ändern. Das hat den Ziel, die Aufgabe des Programmierers zu vereinfachen, weil er soll die gleichzeitigen Datenzugriffs nur für diese Aktivitäten betrachten, die lokal an dem Place sind. Alle Objekte befinden sich in einem bestimmten Place und jedes Objekt hat seine eigenen Place (*home*). Die PGAS Modell erlaubt die Modifikation von Objekte, nur von diese Aktivitäten, die sich lokal auf dem gleichen Ort befinden. Das bedeutet, dass wenn wir einen Objekt verändern wollen, aus einen sich in unterschiedliche Place befindeten Aktivität, müssen wir eine neue Aktivität starten, die sich in den selben Place wie der Objekt befindet. Es gibt auch Methoden und Objekte, die global sind. Die sind von überall zugänglich, aber globale Objekte sollen unbedingt immutable sein und globale Methoden sind schwerer zu entwickeln.



4. Asynchrone Aktivitäten

X10 ermöglicht nicht nur die Aufgaben parallel zwischen verschiedenen Orten zu laufen, sondern auch die gleichzeitige Durchführung von Aktivitäten in den selben Ort. Konkurrente Aktivitäten sind sehr nutzlich für Aufgaben wie IO, weil wir den Resourcen des Rechners benutzen können bis die IO-aufruf fertig ist. Hier ist ein Beispiel wie wir eine Asynchrone Aktivität starten können, mittels einen *async* Block:

```
async head.tap();
async stomach.rub();
```

Diese zwei Anweisungen werden gleichzeitig durchgeführt werden. Wir können sowohl einzelnen Anwesungen auch ganze Blöcke verwenden. Wenn wir möchten eine andere Befehl durchzuführen, nach alle konkurrente Aktivitäten zu Ende gekommen sind, können wir den finish Schlüsselwort *finish* benutzen:

```
finish{
async head.tap();
async stomach.rub();
}
car.drive();
```

In dieser Beispiel ist die dritte Anweisung durchgeführt, nur wenn beide Aktivitäten in den Finish-Block fertig sind. Wenn wir Finish nicht benutzten, könnte die dritte Anweisung zu früh gestartet werden, vor die erste zwei fertig sind.
In Konkurrente Programmen gibt est oft Situationen, wo es einen Resource gibt, auf den nur einen Thread gleichzeitig zugreifen darf. Das kann zum Beizpiel eine Schreib-Operation auf einen gemeinsamen Datei sein. Um einen sicheren Zugriff auf gemeinsamen Resourcen zu haben, können wir die X10 *atomic* Blöcke benutzen:

```
async{
alice.eat();
atomic salt.use();
}
async{
bob.eat();
atomic salt.use();
}
```

Wenn eine Anweisung oder einen Block von der *atomic* Stichwort vorangestellt ist, wird es nicht durch andere Aktivitäten, die die gleichen Ressourcen benutzen, unterbrochen werden. Weil atomaren Blöcken andere Aktivitäten blockieren können, sollten sie mit Vorsicht verwendet werden. X10 führt statische Analyse auf der Source-Code, um Datenabhängigkeiten zu analysieren, so dass eine atomare Block nicht die Ausführung von Aktivitäten blockiert, mit denen er nicht verbunden ist. Einen Method kann auch atomic sein, was das gleiche ist, als wäre den ganzen Körper des Methods in einem atomaren Block geschrieben. X10 erlaubt uns auch einer Ausdruck asynchron zu bewerten mittels future. Das Ergebnis des Ausdrucks kann später mittels den () Operator des Future erhaltet werden:

```
val promise: Future[T] = future (a.dist(pt)) a(pt);
val value: T = promise();
```

Hier der Wert promise ist von der Typ *Future<T>*. Wenn wir den () Operator aufrufen, blockiert der Aufruf bis ein Ergebnis von Typ *T* ausgegeben wird.
Aktivitäten können auch von anderen Places gestarted werden. Auf diese Weise können wir Objekte manipulieren, die sich nicht in den selben Place befinden. Dafür gibt es in X10 das at Schlüsselwort:

```
async at (x.home) x.work();
```

5. X10 für Funktionale Programmierung

In X10 ist es möglich, Daten in *Variablen* oder *Werte* zu speichern. Die Werte sind unveränderlich, d.h. einmal eingestellt, ihr Wert bleibt konstant. Funktionen können auf die gleiche Weise wie Werte definiert werden. Dafür sollen wir nur die Input und Output angeben. Funktionale Programmierung ermöglicht uns, Funktionen als Argumente an andere Funktionen zu benutzen. Wir können auch anonyme Funktionen schreiben:

```
val square = (i:Int) => i*i;
val of4 = (f: (Int)=>Int) => f(4);
val fourSquared = of4(square);
x10.io.Console.OUT.println("4 squared is " + fourSquared);
x10.io.Console.OUT.println("4 cubed is " +
of4( (i:Int)=>i*i*i) );
```

Mögliche Anwendungen für Funktionale Argumente sind zum Beispiel um einen Array zu initialisieren oder eine Reduce-Operation durchzuführen. Es ist auch möglich eine Funktion an einen anderen Ort zu kopieren.

6. Guards

Der X10-Compiler kann verschiedene Eigenschaften des Codes durch eine statische Analyze überprüfen. Der Benutzer kann Nebenbedingungen, die so genannte *Guards*, die eine Eigenschaft eines Objekts oder das Verhalten einer Methode angeben. Ein Guard kann beispielsweise benutzt werden, um die Eingabe Argumente und den Rückgabetyp einer Methode weiter zu spezifizieren, als nur mittels den Typ möglich ist. Zum Beispiel eine Methode, die das Skalarprodukt zweier Vektoren berechnet kann ein Guard haben, der überprüft, ob beide Vektore die gleiche Große haben.

```
static def dot(v: ValRail[Double], w: ValRail[Double] )
{v.length == w.length})= {
    //Method body
}
```

Wir können auch Guards benutzen, um einen mehr spezifischen Typ zu bekommen:

static type Vector3 = ValRail[Double]{length == 3};

Durch die statische Überprüfung der Guards, kann der Compiler potentielle Fehler finden. Das kann Zeit vom Debuggen und Fehlerbehebung sparen.

7. Punkte und Regionen

Ein interessantes Merkmal der Sprache ist, dass Arrays werden von *Punkten* indiziert. Punkten sind n-dimensionalen Tupel von ganzen Zahlen. Zum Beispiel zweidimensionalen Arrays werden durch 2D-Punkten indiziert. X10 unterscheidet sich von den meisten Programmiersprachen in dem, dass beliebige ganze Zahlen die Indizes des Arrays bilden können. Das heißt, wir können Arrays bilden, die ab 1 anstatt 0 oder sogar ab -100 starten. Die Menge der Punkte von dem ersten bis zum letzten Index heißt in X10 eine *Region*. Das ermöglicht uns die Regionen für das Schreiben von Schleifen in einer kurzen Form zu verwenden.

```
for( var (i) in 1..100) {…}
```

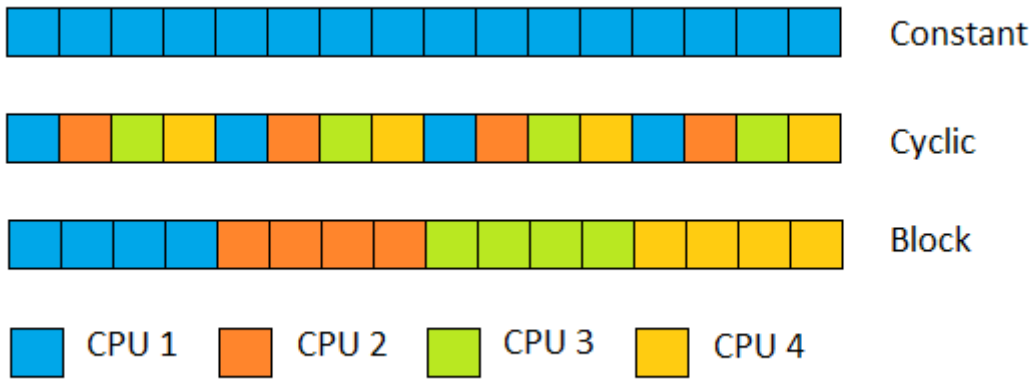
1 .. 100 ist hier die Region der Punkte von 1 bis 100. Beachten Sie, dass *i* eine ganze Zahl ist, aber *(i)* ist ein Punkt im eindimensionalen Raum. Eine Region kann auch n-dimensional sein, in welchem Fall es enthält die Menge der ganzzahlige Punkte im Inneren eine n-dimensionale Figur. Es ist bemerkenswert, dass X10 nicht nur Rechtecke als Form der Region unterstützt, sondern auch beliebige Formen, wie z.B. Dreiecke. Diese Funktion ist zum Beispiel für die Darstellung von Dreiecksmatrizen nützlich. In der aktuellen Version des X10 nicht rechteckigen Regionen sind durch eine rechteckige Begrenzungsrahmen dargestellt, in denen nur einige der Indizes sind benutzbar. Wir können auch neue Regionen mittels Mengenoperationen wie Vereinigung, Durchschnitt oder symmetrische Differenz formen. Zum Beispiel:

```
val U1 : Region{rank==2} = [1..2, 3..4];
val U2 : Region{rank==2} = [100..200, 300..400];
val U3  = U1 || U2; //Union
val U4 = U1 && U2; //Intersection
val U5 = U1 - U2;  //Set difference
val U6 = U1 * U2; //Cartesian product
```

8. Distributions and Arrays

X10 ermöglicht, dass die Daten von einem Array auf verschiedene Orte verteilt werden. Auf diese Weise können wir die Verarbeitung leicht zwischen verschiedenen Rechenknoten verteilen. Es gibt spezielle Strukturen, die so genannten *Distributionen*, die die Information halten, wo sich jedes Element in dem Array befindet. Anders gesagt, ist ein Distribution eine Abbildung von eine Region zu die Menge der Orte. Das bedeutet, dass jedes Element aus der Region wird sich in den Ort befinden, der in die Distriution angegeben ist. Wir können eine Distribution aus einer Region mittels verschiedenen Methoden bilden. Zum Beispiel, wenn wir wollen, dass die Rechenknoten große Stücke von kontinuierlichen Daten bekommen, können wir *Dist.makeBlock()* benutzen. Eine weitere Möglichkeit ist die Verwendung *makeCyclic()* oder *makeRandom()*, z.B. wo Lastverteilung ein Problem ist.

```
val a : Dist = Dist.makeConst(R);
val b : Dist = Dist.makeCyclic(R);
val c : Dist = Dist.makeBlock(R);
```



X10 unterstützt sowohl lokale als auch verteilte Arrays. Um ein verteilten Array zu bekommen, müssen wir eine Distribution an den Konstruktor geben, um zu zeigen, wo sich die Elemente befinden. Wir können auch eine Funktion benutzen, um die Elemente zu initialisieren.

```
val R : Region = 1..35;
val D : Dist = Dist.makeBlock(R);
val f : (Point)=> Int = ((i):Point) => i*i;
val a : Array[Int] = Array.make[Int](D, f);
```

Wir können globale reduce-operationen wie Summe und Produkt benutzen, aber X10 die Möglichkeit bietet, reduce mit beliebige Funktion durchzuführen.

```
a.reduce((i: Int, j: Int)=> i+j , 0); //Sum
```

Wenn wir die Elemente eines Arrays manipulieren wollen, können wir den *at* Schlüsselwort benutzen um eine Neue Aktivität zu starten. Die neue Aktivität wird sich in denselben Ort wie der Element befinden. Wir können auch eine Aktivität für jede Ort starten, die nur die lokale Elemente manipuliert:

```
public static def add(a:Array[Int], b:Array[Int])
{a.dist == b.dist} :Array[Int]{self.dist == a.dist} = {
    c : Array[Int]{dist == a.dist}
         = Array.make[Int](a.dist, (p:Point)=>0);
    val D = a.dist;
    val places : ValRail[Place] = D.places();
    for(place in places) {
        at(place) {
            val pointsAtP : Region{rank == D.rank} =
D.get(place)
            for(pt in pointsAtP)
               c(pt) = a(pt) + b(pt);
        }// at(p)
    }//for
    return c;
}
```

9. Synchronization

In den meisten parallelen Programmen ist es Notwendig, die Durchführung von verschiedenen Aktivitäten zu synchronisieren. Dazu nutzt X10 ein Ansatz, bei dem die Berechnung in Phasen aufteilt ist. In der Phase k arbeiten alle Tätigkeiten parallel, aber sie synchronisieren miteinander, bevor sie zum Nächste Phase gehen. Diese Verallgemeinerung von Barrieren nennt sich in X10 *Clocks*. Wenn eine Aktivität *next* ausführt, signalisiert sie, dass sie bereit für die nächste Phase ist. Die Aktivität dann blockiert bis alle anderen Aktivitäten auch fertig sind. Es ist möglich, nur beliebige Aktivitäten bei einem Clock zu Registrieren. Eine Aktivität kann auch bei mehr als einem Clock registriert werden.

```
val c1: Clock = Clock.make();
...
async clocked (c1,c2,c3) {
...
//Bereit fuer naechste Phase
c1.resume();
...
//Bereit fuer naechste Phase, warte fuer andere Activities
c1.next();
...
c2.drop()
}
```

Die X10 Sprache hat eine Menge von Regeln, die festlegen, wie Clocks gebraucht werden sollen. Diese Regeln sind bei der Kompilierung überprüft und garantieren, dass keine Deadlocks auftreten.

10. Fazit

Obwohl noch jung, X10 ist eine sehr fortgeschrittene Programmiersprache. Die sequentiellen Funktionen geben die Sprache Flexibilität um komplexe Programme zu schreiben. Mit X10 ist das schreiben von parallelem Code viel einfacher als den meisten anderen Sprachen.

11. Literatur

- Report on the Programming Language X10. Vijay Saraswat, Bard Bloom
- Kemal Ebcioglu, Vijay Saraswat, Vivek Sarkar. X10: an Experimental Language for High Productivity Programming of Scalable Systems. P-PHEC workshop, HPCA 2005.
- Philip Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, Vivek Sarkar. X10: An Object-oriented approach to non-uniform Clustered Computing. OOPSLA 2005.
- Philip Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, Vivek Sarkar. X10: An Object-oriented approach to non-uniform Clustered Computing. OOPSLA 2005.
- Ganesh Bikshandi, José G. Castaños, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, Tong Wen: Efficient, portable implementation of asynchronous multi-place programs . PPOPP 2009: 271-282
- Guojing Cong, Gheorghe Almasi, Vijay Saraswat:Fast PGAS Connected Components Algorithms . Presented at PGAS conference, 2009.
- Deadlock-Free Scheduling of X10 Computations with Bounded Resources. Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna Shyamasundar, Katherine Yelick. Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA ’07), June 2007.