

UNIVERSITÄT PASSAU  
Fakultät für Informatik und Mathematik

## **X10**

im Hauptseminar “Multicore Programming”

Sebastian Henneberg  
Universität Passau, 8. Dezember 2011

# Inhaltsverzeichnis

<b>1</b>	<b>Hintergrund</b>	<b>2</b>
1.1	Geschichte . . . . .	2
1.2	Entwicklung . . . . .	2
1.3	Inspiration . . . . .	2
1.4	Motivation . . . . .	2
<b>2</b>	<b>Plattform</b>	<b>3</b>
2.1	Typsystem . . . . .	3
2.2	Paradigmen . . . . .	3
2.3	Syntaktische Nähe zu Java . . . . .	3
2.4	Backends . . . . .	4
<b>3</b>	<b>Speicher- und Parallelitätsmodell</b>	<b>5</b>
3.1	Speichermodell . . . . .	5
3.2	Aktivitätsmodell . . . . .	6
<b>4</b>	<b>Sprache</b>	<b>6</b>
4.1	Sprachkonstrukte . . . . .	7
4.1.1	<code>async</code> -Schlüsselwort . . . . .	7
4.1.2	<code>finish</code> -Schlüsselwort . . . . .	7
4.1.3	cilk-style Fork/Join . . . . .	7
4.1.4	<code>at</code> -Schlüsselwort . . . . .	8
4.1.5	<code>atomic</code> -Schlüsselwort . . . . .	9
4.1.6	<code>when</code> -Schlüsselwort . . . . .	10
4.1.7	<code>clock</code> -Operationen . . . . .	10
4.2	Evolution von MontyPi . . . . .	10
4.2.1	sequentiell . . . . .	11
4.2.2	nebenläufig . . . . .	11
4.2.3	verteilt . . . . .	12
<b>5</b>	<b>Zusammenfassung</b>	<b>12</b>

# Abbildungsverzeichnis

1	Beispiel für die syntaktische Nähe zu Java . . . . .	4
2	Anwendungskette der C++/Java Backends . . . . .	4
3	Speichermodelle von MPI, OpenMP und X10 (PGAS) . . . . .	5
4	Exemplarischer Aktivitätsbaum eines X10-Programms . . . . .	6
5	kaskadenrekursive Implementierung der Fibonacci-Zahlen . . . . .	8
6	Wechsel der Place durch das <code>at</code> -Statement . . . . .	8
7	Sichtbarkeit und Zugriffsmuster von Variablen in umgebenden Blöcken . . . . .	9
8	Berechnungsverfahren für $\pi$ durch Monte-Carlo-Algorithmus . . . . .	11
9	Die sequentielle Implementierung von MontyPi . . . . .	11
10	Die nebenläufige Implementierung von MontyPi . . . . .	12
11	Die verteilte Implementierung von MontyPi . . . . .	13

# 1 Hintergrund

## 1.1 Geschichte

Die Programmiersprache X10 und die daran gekoppelte Plattform werden seit 2004 erforscht. Erklärtes Ziel des Projekts ist eine Sprache für parallele und verteilte Programmierung zu entwickeln [CGS<sup>+</sup>05]. Der Ursprung des Projekts liegt im Watson Research Center welches von IBM betrieben wird. Die Entwicklung wird durch das HPCS-Programm (High Productivity Computing Systems) der DARPA unterstützt. Das Ziel dieses Programms ist die Erforschung von kosteneffizienten und hochproduktiven “Petascale”-Systemen für die nationale Sicherheit, die Wissenschaft und die Wirtschaft<sup>1</sup>.

## 1.2 Entwicklung

Aufgrund der wachsenden Anzahl an Ressourcen und Referenzen kann von einem steigenden Interesse an der Programmiersprache X10 ausgegangen werden. Besonders die Anzahl der Publikation mit Bezug auf X10<sup>2</sup> ist in den letzten Jahren massiv angestiegen. Außerdem wurden in den Jahren 2010 und 2011 erstmals Workshops angeboten. Der Interessentenkreis scheint jedoch sehr stark im akademischen Umfeld angesiedelt zu sein.

Im Jahr 2009 gewann X10 den Preis *Best Performance*<sup>3</sup> der *HPC Challenge*, welche ebenfalls durch das HPCS-Programm der DARPA gefördert wird. X10 erhielt den Preis aufgrund der Kombination von Performanz, Länge des Programmtextes, Lesbarkeit und benötigtem Programmieraufwand.

## 1.3 Inspiration

Als Inspiration für X10 dienen verschiedene Projekte und Programmiersprachen. Da X10 unter Anderem ein Java-Backend besitzt, kann das mittlerweile verwaiste Projekt *Java Party* [PZ97] als Inspiration angesehen werden. Das Projekt bietet eine vergleichsweise einfache Schnittstelle, welche einen Verbund von JVMs (Java Virtual Machine) zusammen stellt um rechenintensive Operationen verteilt auszuführen. Das Speichermodell PGAS (Partitioned Global Address Space) sowie Ausführungsmodell von X10 ist eine Mischung aus MPI (Message Passing Interface) und OpenMP (Open Shared-Memory). Beides sind Konzepte/Frameworks für die Cluster-Programmierung mit *C*, *C++* oder Fortran. Das Speichermodell von X10 wurde zuvor bereits im Entwurf anderer Programmiersprachen wie *Unified Parallel C*, *Co-array Fortran*, *Fortress* und *Chapel* verwendet.

## 1.4 Motivation

Die Motivation für die X10-Plattform basiert vor allem in einer zugänglichen Schnittstelle für die Cluster-Programmierung um die Produktivität im Entwicklungsprozess deutlich zu steigern. Dies soll durch eine Java-ähnliche Syntax erreicht werden. Die Sprache selbst ist mit zahlreichen Konstrukten ausgestattet um parallele und nebenläufige Aufgaben leicht implementieren zu können. Ein weiteres Ziel der Plattform ist eine hohe Geschwindigkeit die durch verschiedene Backends (*C++* und *Java*) erreicht werden soll. Zur Geschwindigkeitssteigerung kann können verschiedene dokumentierte Schalter und Techniken<sup>4</sup> verwendet werden. Die X10-Plattform hat ihren Namen der angesprochenen Geschwindigkeit zu verdanken. “X10” steht nämlich für  $\times 10$  und bedeutet “10 mal schneller”. Des Weiteren spielt die Skalierung eine zentrale Rolle. X10-Programme laufen sowohl auf Einkern- als auch auf Mehrkern-Prozessoren. Primäre Zielplattformen sind allerdings Cluster und Supercomputer.

---

<sup>1</sup><http://archive.hpcwire.com/hpc/1119092.html>

<sup>2</sup><http://x10-lang.org/x10-community/publications-using-x10.html>

<sup>3</sup><http://www.hpcchallenge.org/custom/index.html?lid=103&slid=236>

<sup>4</sup><http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html>

## 2 Plattform

### 2.1 Typsystem

X10 ist eine statisch und streng getypte Programmiersprache. Typdeklarationen sind nicht an jeder Stelle verpflichtend anzugeben. Durch Typinferenz kann der Typ einer Variablen vom Compiler ermittelt werden. Die Typinferenz findet jedoch nur durch die initialisierende Deklaration statt. X10 besitzt im Gegensatz zu C und C++ keine Zeiger. Stattdessen arbeitet man mit Referenzen. Neben zahlreichen - aus `java.util` bekannten - Collections, können eigene Typen durch Klassen, Schnittstellen oder Strukturen (`struct`) definiert werden.

### 2.2 Paradigmen

Die Sprache X10 besitzt kein einheitliches Paradigma. Vielmehr bietet die Sprache eine Kombination aus nützlichen Eigenschaften sehr verschiedener Paradigmen. Da viele Statements mit Seiteneffekten behaftet sind, handelt es sich auf jeden Fall um eine imperative Programmiersprache. Die objektorientierte Komponente wird durch die Definition von Klasse und Schnittstellen und dem Vererbungs-Prinzip dargestellt. Die nebenläufige Komponente wird durch die Plattform selbst und zahlreiche Sprachkonstrukte repräsentiert. Dabei spielen die Eigenschaften *Lokalität*, *Asynchronität*, *Atomarität* und *Ordnung* zentrale Rollen. Aufgrund der angesprochenen Skalierung auf Clustern kann man X10 auch als verteilte Programmiersprache ansehen. Die Unterstützung von verteilter Programmierung wird durch *Message Passing* und das angepasste Speichermodell erreicht. Da es in X10 zusätzlich möglich ist, *Closures* zu erzeugen, kann man X10 auch als funktionale Sprache verstehen. Weitere funktionale Eigenschaften sind *Immutable State* und *First-Class Functions*. Letztere erlauben Funktionenobjekte als Parameter an andere Funktionen zu übergeben oder von anderen Funktionen als Rückgabewert zu erhalten. Der Schwerpunkt der Sprache liegt jedoch bei der **verteilten nebenläufigen** Programmierung.

### 2.3 Syntaktische Nähe zu Java

Die Sprache X10 lehnt sich an vielen Stellen an der Java-Syntax an. Somit fällt der Einstieg in die Sprache für Java-Entwickler leicht. X10 enthält einen reichen Wortschatz an Schlüsselwörter für nebenläufige und verteilte Umgebungen (siehe Kapitel 4). Die allgemeine Syntax wurde an einigen Stellen angepasst um bestimmte Eigenschaften der Sprache zu realisieren, die im vorherigen Abschnitt aufgezählt wurden.

Unabhängig von den speziellen Parallelitätseigenschaften betrachten wir den Programmtext der `Factorial`-Klasse in Abbildung 1. Dieses sequentielle Programm berechnet abhängig von einer Benutzereingabe die Fakultät einer Ganzzahl  $n$ . Im folgenden wollen wir auf Unterschiede zur Java-Syntax aufmerksam machen und diese erläutern:

- Das aus Python oder Scala bekannte `def`-Schlüsselwort leitet Funktionen bzw. Methoden ein.
- Rückgabetypen einer Funktion bzw. Methode werden erst nach der Argumentaufzählung und beginnend mit einem Doppelpunkt eingeleitet.
- Datentypen der Argumente einer Funktion bzw. Methode werden ebenfalls erst nach dem Argumentnamen und beginnend mit einem Doppelpunkt notiert.
- Variablen sind entweder `vars` (veränderbar) oder `vals` (unveränderbar). Variablen die mit einem `val` deklariert sind können sich also im Java-Kontext als `final` verstehen.
- Alle Typen in X10 beginnen mit einem großen Buchstaben. Neben den aus Java bekannten Typen gibt vorzeichenlose Varianten der Ganzzahl-Typen `Byte`, `Short`, `Int` und `Long`. Diese sind durch ein großes "U" vor dem eigentlichen Typnamen gekennzeichnet.

- for-Schleifen lassen sich durch das `in`-Schlüsselwort einfach auf Aufzählungen (`IntRange` oder `LongRange`) anwenden.

```

1 public class Factorial {
2
3     public static def main(args: Array[String]) {
4         val n = Int.parse(args(0));
5         Console.OUT.println(n + "! = " + fac(n));
6     }
7
8     private static def fac(n: Int): ULong {
9         var f : ULong = 1;
10        for (i in 1..n) {
11            f *= i;
12        }
13        return f;
14    }
15 }

```

Abbildung 1: Beispiel für die syntaktische Nähe zu Java

## 2.4 Backends

X10 besitzt zwei primäre Backends. Diese unterscheiden sich bereits dahingehend, wie der Programmtext kompiliert werden soll. Das C++-Backend ermöglicht nativen Code für eine spezielle Zielplattform zu erzeugen. Nachdem das Programm mittels `x10c++` übersetzt wurde, kann es mit `runx10` ausgeführt werden. Das Java-Backend erzeugt durch Anwendung von `x10c` gewöhnlichen JVM-Bytecode. Dieser wird jedoch gekapselt über das Programm `x10` zur Ausführung gebracht. Eine direkte Ausführung mittels `./` beim nativen Programm oder mittels `Java` beim JVM-Bytecode ist nicht möglich, da die X10-Plattform einige Vorbereitungen für die verteilte und nebenläufige Umgebung vornehmen muss. Die Anwendungskette der angesprochenen Werkzeuge abhängig vom verwendeten Backend ist in Abbildung 2 visualisiert.

Momentan ist das C++-Backend schneller und unterstützt mehr Kommunikationsprotokolle. In Zukunft soll die Interoperabilität beider Backends zu verschiedenen Plattformen ausgebaut werden. Dabei soll das Java-Backend mit reinen Java-Programmen arbeiten können und das C++-Backend mit verschiedenen Bibliotheken für Hardware-Beschleunigung und GPUs. Möglicherweise werden die Backends sogar kompatibel zueinander gemacht um bestimmte Teile auf der JVM auszuführen und andere nativ<sup>5</sup>.

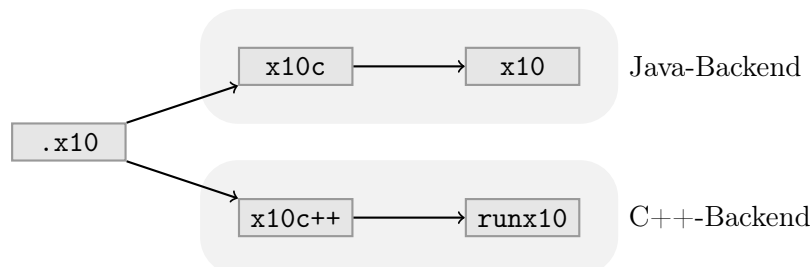


Abbildung 2: Anwendungskette der C++/Java Backends

<sup>5</sup><http://x10-lang.org/home/faq-list.html>

### 3 Speicher- und Parallelitätsmodell

Die X10-Plattform unterscheidet sich vor allem durch ein besonderes Speichermodell, welches nur in sehr wenigen anderen Programmiersprachen vorliegt. Das Design dieses Speichermodells hat direkten Einfluss auf den Kontrollfluss, welcher in X10 in sog. Aktivitäten erfolgt. Dieses Kapitel erläutert die Kern-Konzepte der X10-Plattform die durch das Speichermodell und die Aktivitäten charakterisiert sind.

#### 3.1 Speichermodell

Das in X10 vorliegende Speichermodell heißt APGAS [SAB<sup>+</sup>10] (Asynchronous Partitioned Global Address Space) und basiert auf dem Prinzip der Lokalität. Dazu werden zu Ausführungsbeginn *Places* erzeugt, welche jeweils den Speicherort eines beliebigen Objekts gemäß des Lokali-tätsprinzips ausdrücken. Der global verfügbare Speicher wird dazu partitioniert und anschließend den vorhandenen Places exklusiv zugeordnet. Die Instanzierung von Datenobjekten innerhalb einer Place bindet die Zugriffsbeschränkung für die gesamte Lebensdauer an das jeweilige Place. Die Nutzung von Datenobjekten außerhalb der Erzeuger-Place kann durch verschiedene Techniken erreicht werden auf die im Abschnitt 4.1.4 genauer eingegangen wird. Das PGAS-Modell wurde bereits in den Programmiersprachen Unified Parallel C (UPC), Fortran, Fortress und Chapel verwendet.

Um das Speichermodell von X10 besser zu verstehen, betrachten wir das MPI-Framework<sup>6</sup> (Message Passing Interface) und die OpenMP-API<sup>7</sup> (Open Multi-Processing). Die Einflüsse beider Modelle bilden die Inspiration für das PGAS-Modell und insbesondere X10. Abbildung 3 zeigt alle drei erwähnten Speichermodelle im Vergleich. MPI bedient sich des Message Passing um Datenobjekte zu serialisieren und per Nachricht an asynchrone Threads zu senden. Durch den Empfang am Ziel-Thread können die Datenobjekte nach der Deserialisierung verwendet werden. Im MPI-Modell ist kein gemeinsamer Speicher vorgesehen, weshalb die Datenobjekte vor der Serialisierung kopiert werden müssen. OpenMP hingegen macht sich explizit den gemeinsamen Speicher zu nutze um solche kostenintensiven Kopien zu vermeiden. Dazu bietet OpenMP eine Fülle von Compiler-Direktiven und Bibliotheken an um die parallele Ausführung auf gemeinsamen Speicher zu ermöglichen. Das PGAS-Modell stellt eine Mischform dar, weil die Speicherbereiche zwar exklusiv verteilt, jedoch trotzdem auf Datenobjekte anderer Places zugegriffen werden kann. Darüberhinaus können Datenobjekte wie z.B. Arrays auch über mehrere Places hinweg verteilt sein.

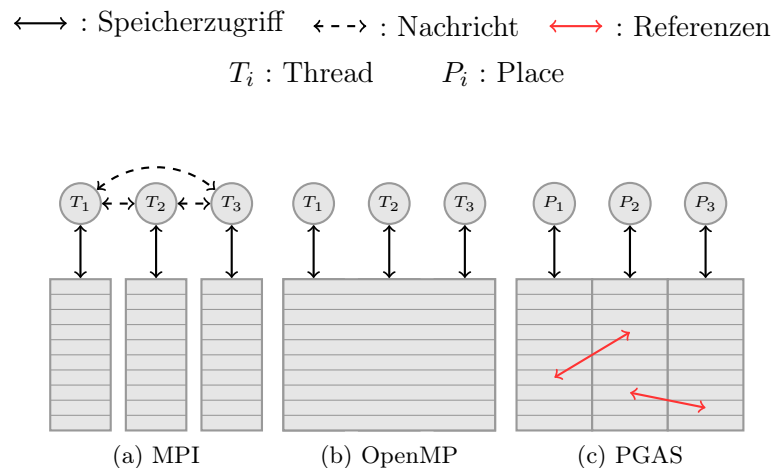


Abbildung 3: Speichermodelle von MPI, OpenMP und X10 (PGAS)

<sup>6</sup><http://www.mcs.anl.gov/research/projects/mpi/>

<sup>7</sup><http://openmp.org/wp/>

### 3.2 Aktivitätsmodell

Die Ausführung von Statements und Routinen innerhalb von X10 findet in *Aktivitäten* statt. Aktivitäten bilden somit die Ausführungsstränge welche der Abarbeitung des Programms dienen. Aktivitäten operieren im Allgemeinen unabhängig voneinander und somit vollkommen asynchron. Neue Aktivitäten können lediglich mittels des `async`-Statements erzeugt werden. Jede Aktivität ist immer genau einer Place zugeordnet. Durch Benutzung des `at`-Statements kann eine Aktivität zu einer anderen Place transportiert werden. Die Anzahl der Aktivitäten die sich an der gleichen Place befinden ist nicht beschränkt. Jedoch sollte sicher gestellt sein, dass die Aktivitäten sich gegenseitig nicht behindern. Andernfalls können *Race Conditions* auftreten. Aktivitäten in X10 können als sehr leichtgewichtige Threads verstanden werden.

Da zu Programmstart nur eine Aktivität existiert, müssen weitere Aktivitäten dynamisch per `async` erzeugt werden. Dies resultiert in einem Aktivitätsbaum wie in Abbildung 4. Die initiale Aktivität, auch *root activity* genannt, startet in Place 0 in der `main`-Methode. Kurz darauf wird mit `async` eine neue Aktivität erzeugt und mittels `at` direkt zu Place 2 transportiert. Die Anwendung von `async` und `at` kann auch unabhängig zu verschiedenen Zeitpunkten stattfinden, wie man im späteren Verlauf an Place 1 erkennen kann. Mithilfe des `finish`-Statements kann man Synchronisationspunkte erzeugen, welche auf die Abarbeitung aller erzeugten Aktivitäten des Unterbaums warten. Für das Beenden einer Aktivität ist lediglich die Aktivität selbst verantwortlich. Im Beispiel endet eine Aktivität an Place 1 kurz vor Ende des betrachteten Aktivitätsbaums.

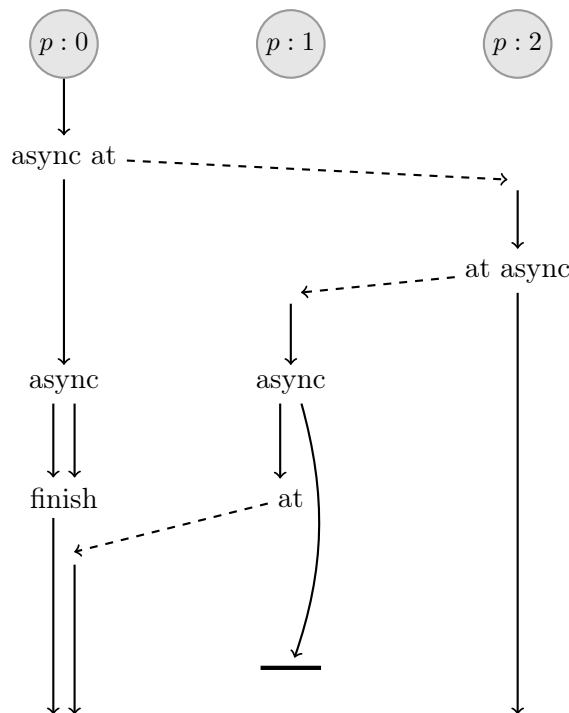


Abbildung 4: Exemplarischer Aktivitätsbaum eines X10-Programms

## 4 Sprache

Dieses Kapitel gliedert sich in zwei Teile. Der erste Teil befasst sich formal und theoretisch mit den Sprachkonstrukten in X10. Im zweiten Teil werden die vorher eingeführten Sprachkonstrukte schrittweise in ein sequentielles Programm eingearbeitet um schlussendlich ein nebenläufiges und

ein verteiltes Programm zu erzeugen.

## 4.1 Sprachkonstrukte

Trotz starker Anlehnung an die Java-Syntax, führt X10 zahlreiche neue Statements und Schlüsselwörter ein, um eine ideale Plattform für verteilte und nebenläufige Anwendungen darzustellen. Die wichtigsten Konstrukte werden in diesem Kapitel nacheinander eingeführt und detailliert beschrieben. Zur Orientierung und zur Erleichterung des Verständnisses werden äquivalente oder ähnliche Konzepte aus Java kurz angesprochen.

### 4.1.1 `async`-Schlüsselwort

**`async`** { `S` }

Das `async`-Statement erzeugt eine neue Aktivität welche `S` ausführt. Der Kontrollfluss der aufrufenden Aktivität kehrt dabei sofort zurück und erzeugt eine neue asynchrone Aktivität an der gleichen Place.

Aktivitäten in X10 können nicht abgebrochen werden. Jede Aktivität muss selbstständig die Berechnung abschließen. Variablen in umgebenden Blöcken können referenziert werden. Jedoch nur wenn diese mit `val` eingeführt wurden. Falls die ursprüngliche Aktivität in der Zwischenzeit beendet wurde, müssen alle in `S` referenzierten Variablen weiterhin lebendig bleiben. Dabei handelt es sich um ein Szenario, welches von Closures bereits bekannt ist. Aufgrund der asynchronen Ausführung können `break` und `continue` in `S` nicht auf Schleifen im umgebenden Block angewendet werden.

Das `async`-Statement sollte nicht wie ein `Thread.start()` aus Java verstanden werden. Aufgrund der unterschiedlichen Speichermodelle lässt sich die größte Gemeinsamkeit lediglich von der asynchronen Ausführung ableiten.

### 4.1.2 `finish`-Schlüsselwort

**`finish`** { `S` }

Das `finish`-Statement bietet eine einfache Möglichkeit asynchrone Aktivitäten zu synchronisieren. Dazu werden alle (transitiv) in `S` erzeugten Aktivitäten gesammelt und beobachtet. Sobald all diese Aktivitäten ihre Ausführung beendet haben, wird die Ausführung in hinter dem `finish`-Block fortgeführt.

Diese einfache Synchronisation von asynchron ausgeführten Kontrollflüssen wurde in Java 6 unter dem Namen des `CountDownLatch` eingeführt. Dieser ermöglicht das Warten eines Threads bis eine zuvor spezifizierte Zahl an Threads ein Signal an den `CountDownLatch` gegeben haben.

### 4.1.3 `cilk-style Fork/Join`

Da Synchronisation und das Erzeugen von Aktivitäten direkt voneinander abhängig sind, treten `finish` und `async` i.d.R. immer in Kombination miteinander auf. Dies wurde bereits im Aktivitätsbaum in Abbildung 4 gezeigt. Das durch `async` und `finish` konstruierte Berechnungsschema ist auch unter dem Namen *Fork/Join* bekannt. Es beschreibt das Aufspalten eines Problems in eine Menge von Teilproblemen die unabhängig von unterschiedlichen Prozessen gelöst werden. Nachdem die Ausführung aller Prozesse erfolgreich beendet ist, können die Ergebnisse der einzelnen Teilergebnisse zum Gesamtergebnis zusammen gebaut werden.

In Abbildung 5a sieht man eine *Fork/Join*-Implementierung der Fibonacci-Zahlen in X10 implementiert. Durch das `finish`-Schlüsselwort wird die Ausführung der Hauptaktivität angehalten bis die Ergebnisse der Funktionsaufrufe von `fib (n-1)` und `fib (n-2)` verfügbar sind. Analog dazu kann man in Abbildung 5b eine äquivalente Implementierung in der Programmiersprache *Cilk* sehen. Das Schlüsselwort `spawn` verhält sich wie `async` und `sync` wie `finish`. Das *Fork/Join*-Berechnungsmodell aus Cilk [FLR98] war eine maßgebliche Inspiration für X10.

In Java 7 wurde ein eigenes generisches Fork/Join-Framework im Paket `java.util.concurrent`

<pre> 1  def fib(n:Int):Int { 2    if (n &lt; 2) return 1; 3    val x:Int; 4    val y:Int; 5    finish { 6      async x = fib(n-1); 7      async y = fib(n-2); 8    } 9    return x+y; 10 } </pre>	<pre> 1  cilk int fib(int n) { 2    if (n &lt; 2) return 1; 3    else { 4      int x,y; 5      x = spawn fib (n-1); 6      y = spawn fib (n-2); 7      sync; 8      return x+y; 9    } 10 } </pre>
(a) Fibonacci in X10	(b) Fibonacci in Cilk

Abbildung 5: kaskadenrekursive Implementierung der Fibonacci-Zahlen

eingeführt. Mit dessen Hilfe lassen sich beliebige Fork/Join-Aufgaben auch in Java leicht implementieren.

#### 4.1.4 at-Schlüsselwort

`at(p) { S }`

Das `at`-Schlüsselwort dient zum synchronen Wechsel der Aktivität zu einer anderen Place `p`. Die Statements in `S` werden dann in `p` ausgeführt. Nach Ausführung von `S` kehrt die Aktivität zur ursprünglichen Place zurück. Abbildung 6 beschreibt dieses Verhalten durch ein Sequenz-Diagramm.

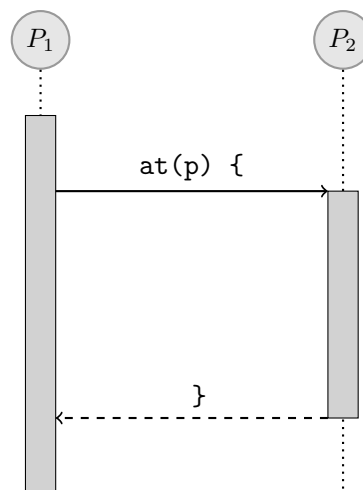


Abbildung 6: Wechsel der Place durch das `at`-Statement

Der Wechsel zu einer anderen Place stellt die Basis für verteilte X10-Programme dar. Allerdings kann der Wechsel dazu führen, dass die Aktivität nicht nur den lokalen Speicherbereich einer Maschine wechselt, sondern an eine komplett andere Maschine transportiert wird. Aus diesem Grund kann die Benutzung von `at` sehr teuer sein, da im Hintergrund ein Message Passing über das Netzwerk stattfindet.

Das `at`-Statement erlaubt sogar Variablen in `S` zu referenzieren, die in umgebenden Blöcken definiert bzw. benutzt wurden. Dabei gibt es allerdings ein paar Dinge zu beachten:

- `var`-Variablen sind in **S** nicht sichtbar.
- `val`-Werte sind als Kopie in **S** verfügbar.
  - Für die Kopien werden *Deep Copies* von den Datenobjekten erzeugt.
  - Alle als `transient` markierten Variablen werden nicht kopiert. Bei solchen Variablen wird der Initialisierungswert benutzt.
  - Falls die Variable vom Typ `GlobalRef[T]` ist, wird ebenfalls keine Kopie erstellt. Variablen von diesem Typ sind in anderen Places sichtbar. Jedoch kann nur in der Place, die das `GlobalRef[T]`-Objekt erzeugt hat, auf den Inhalt zugegriffen werden.

Damit die benötigten Variablen an der Place **p** nach obiger Semantik verfügbar sind, müssen Kopien oder Referenzen an die Place **p** gesendet werden. Dazu werden alle in **S** referenzierten Datenobjekte serialisiert, in einen Buffer geschrieben und an **p** gesendet. An der Place **p** wird der Buffer ausgelesen und die Datenobjekte werden deserialisiert, damit darauf operiert werden kann. Abbildung 7 visualisiert die Sichtbarkeit sowie die Zugriffsmethoden auf die äußeren Datenobjekte. Auf **i** kann innerhalb des `at`-Blocks nicht zugegriffen werden, da **i** mit `var` eingeführt wurde. Die mit `val` eingeführten Werte **c** und **r** hingegen sind innerhalb des `at`-Blocks sichtbar. Da `val`-Werte unveränderlich sind, benutzen wir die Klasse `Cell` als Wrapper für einen Integer. Die `Cell`-Klasse ermöglicht mittels überladener Operatoren den Zugriff auf das gekapselte Objekt durch die Anwendung von `()`. In der Implementierung inkrementieren wir den Wert von **c** um eins. Da es sich bei **c** im `at`-Block um eine Kopie handelt, ist die Inkrementierung im umliegenden Block zu keiner Zeit sichtbar. Etwas anders verhält es sich mit **r** vom Typ `GlobalRef[Cell[Int]]`. Aufgrund des Typs `GlobalRef[T]` ist **r** in allen Places lesbar. Die überladene Funktion `()` ist jedoch nur aus der Place aufrufbar, in der **r** erzeugt wurde. Aus diesem Grund wird in Zeile 6 temporär das Place gewechselt um den gekapselten Ganzzahlwert in **r** zu inkrementieren. Die Inkrementation von **r** ist in diesem Fall global, da keine Kopie erzeugt wurde.

```

1 var i : Int = 0;
2 val c = new Cell[Int](2);
3 val r = GlobalRef[Cell[Int]](new Cell[Int](4));
4 at(here.next()) {
5     c()++; // sets to 3
6     at(r) { r()()++; } // sets to 5
7 }
8 Console.OUT.println(c()); // prints 2
9 Console.OUT.println(r()()); // prints 5

```

Abbildung 7: Sichtbarkeit und Zugriffsmuster von Variablen in umgebenden Blöcken

#### 4.1.5 atomic-Schlüsselwort

`atomic { S }`

Das `atomic`-Schlüsselwort in X10 bietet die Möglichkeit eine Reihe von Statements atomar auszuführen. Während der Ausführung eines atomaren Blocks kann kein anderer atomarer Block ausgeführt werden, da atomare Blocks in serialisierter Ordnung verarbeitet werden. Dafür sorgt ein spezieller Scheduler in X10. Jedoch gibt es einige Dinge bei der Benutzung von atomaren Blocks zu beachten:

- Innerhalb von **S** darf es keine blockierenden Operationen geben (`when`, `next`, `finish`, ...)

- Innerhalb von **S** darf keine Aktivitäten erzeugen (kein `async`)
- Innerhalb von **S** darf nur auf lokalen Datenobjekten operieren (kein `at`)

Die genannten Einschränkungen werden zur Laufzeit geprüft. Zusammenfassend lässt sich empfehlen atomare Blocks so kurz und unkompliziert wie möglich zu definieren.

Da atomare Blocks in serialisierter Ordnung abgearbeitet werden, können diese gegebenenfalls einen Flaschenhals darstellen. Dies wäre der Fall wenn viele der Blocks überhaupt nicht in Konflikt miteinander stehen. In diesem Fall sind die `synchronized`-Modifier/Statements aus Java deutlich feingranularer, da sie sich nur auf eine konkrete Instanz beziehen.

#### 4.1.6 when-Schlüsselwort

```
when(e) { S }
```

Das `when`-Schlüsselwort blockiert die Aktivität bis das Prädikat `e` als wahr ausgewertet werden kann. Anschließend wird `S` atomar ausgeführt. Die in `e` verwendeten Variablen können jederzeit verändert werden. Daher geschieht die Überprüfung, ob das Prädikat `e` zu wahr ausgewertet werden kann, über *Busy Waiting*. Allerdings empfiehlt es sich die in `e` verwendeten Variablen innerhalb eines atomaren Blocks zu ändern, da anschließend alle blockierenden `when`-Statements benachrichtigt werden.

Da `when` einen atomaren Block beschreibt gilt `when (true) ≡ atomic`.

#### 4.1.7 clock-Operationen

```
val c : Clock = Clock.make();
async clocked (c) { S }
```

Clocks ermöglichen phasenweise Ausführung (Phased Computation) in mehreren asynchronen Aktivitäten. Dazu werden Clocks mit der jeweiligen Aktivität registriert. Dies geschieht während der Erzeugung einer Aktivität über `clocked (c)`. Im Allgemeinen wird eine Clock `c` mehreren Aktivitäten zugeordnet. Durch den Aufruf `c.resumeAll()` in einer der registrierten Aktivitäten erfolgt eine Blockierung der Ausführung. Sobald die letzte registrierte Aktivität diese Methode aufruft, kehren alle Aufrufe zurück und die Ausführung wird in allen Aktivitäten fortgesetzt. Durch dieses Schema können beliebig viele Berechnungsphasen parallel ausgeführt werden.

Java besitzt seit Version 6 die Klasse `CyclicBarrier`. Diese verhält sich vollkommen analog zu Clocks in X10. Lediglich die Registrierung mit den beteiligten Threads muss selbstständig vorgenommen werden.

## 4.2 Evolution von MontyPi

Um die theoretisch erläuterten Konstrukte in X10 besser zu verstehen, werden diese nun schrittweise in ein Beispielprogramm eingeführt. Das Beispielprogramm heißt *MontyPi* und dient dazu die Kreiszahl  $\pi$  anzunähern. Dies geschieht durch Anwendung eines Monte-Carlo-Algorithmus. Abbildung 8a zeigt einen Einheitskreis mit dem Flächeninhalt von  $1^2 \cdot \pi = \pi$ . Zur Vereinfachung des Algorithmus betrachten wir nur ein Viertel dieses Kreises. O.B.d.A. wählen wir dazu den ersten Quadranten wie in Abbildung 8b zu sehen. Innerhalb dieser quadratischen Fläche wählen wir nun über eine gleich-verteilte Zufallsfunktion  $n$  Punkte aus. Sei  $p_i \in (x, y)$  für  $i \in \{1, \dots, n\}$  mit  $x, y \in [0, 1[$ . Für jeden Punkt  $p_i$  prüfen wir, ob dieser innerhalb der grünen Fläche liegt. Dies wird über die euklidische Norm ermittelt. Sei dazu  $h = \#\{p_i \mid \|p_i\| \leq 1\}$ . Der Quotient  $\frac{h}{n}$  gibt nun an zu welcher Wahrscheinlichkeit ein Punkt  $p_i$  in der grünen Fläche liegt. Zudem nähert der Quotient den Flächeninhalt der grünen Fläche an. Um einen angenäherten Wert für  $\pi$  zu erhalten gilt nun folglich  $\pi \approx 4 \cdot \frac{h}{n}$ .

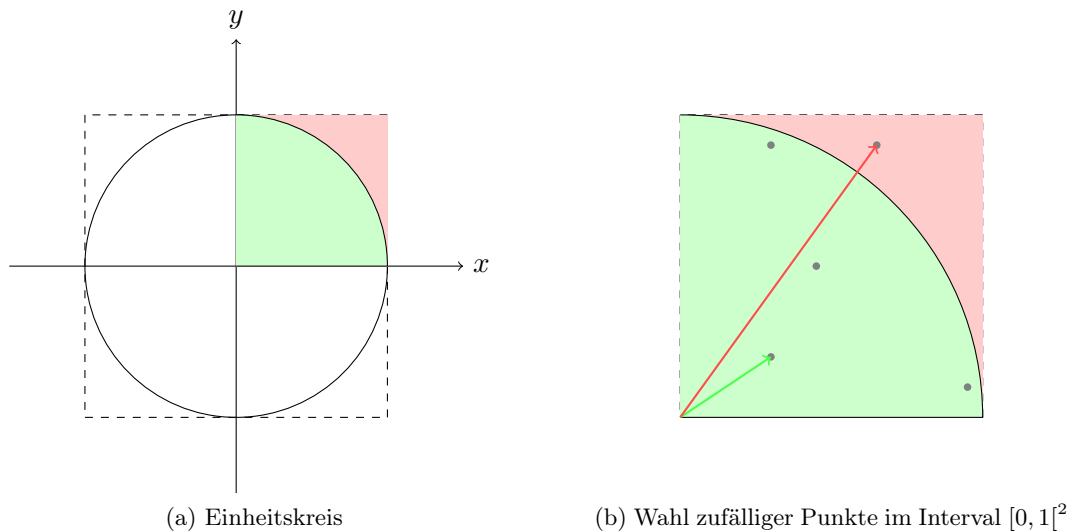


Abbildung 8: Berechnungsverfahren für  $\pi$  durch Monte-Carlo-Algorithmus

#### 4.2.1 sequentiell

Die erste Implementierung in Abbildung 9 dient als Basis für die geplanten Erweiterungen. Die Implementierung ist komplett sequentiell und ermöglicht keine parallele Ausführung. Die Anzahl  $n$  der zufällig zu ermittelnden Punkte wird als Zeichenkette an das Programm übergeben. Anschließend wird ein Zufallsobjekt `R` erzeugt, mit dem die  $x$ - und  $y$ -Koordinaten generiert werden. Nachdem das Gesamtergebnis `result` initialisiert wurde, werden mithilfe einer `for`-Schleife  $n$  zufällige Punkte generiert. Abhängig von der Norm der Punkte wird der `result`-Zähler erhöht. Nach erfolgreicher Abarbeitung der Schleife wird  $\pi$  berechnet und in der folgenden Zeile ausgegeben.

```

4 public static def main(args: Array[String]) {
5     val N = Int.parse(args(0));
6     val R = new Random();
7     var result:Double = 0;
8     for (1..N) {
9         val x = R.nextDouble();
10        val y = R.nextDouble();
11        if (x*x + y*y <= 1) result++;
12    }
13    val pi = 4*result/N;
14    Console.OUT.println("The value of pi is " + pi);
15 }

```

Abbildung 9: Die sequentielle Implementierung von MontyPi

#### 4.2.2 nebenläufig

Die zweite Implementierung in Abbildung 10 führt einige Erweiterungen ein um eine nebenläufige Ausführung zu ermöglichen. Der Kern der nebenläufigen Ausführung ist nahezu komplett identisch zum sequentiellen Programm. Er besteht aus der `for`-Schleife und der Initialisierung von `R` und `result`. Lediglich die obere Grenze der `for`-Schleife ist angepasst, da die Generierung von zufälligen Punkten nun auf mehrere Aktivitäten aufgeteilt ist.

Im Gegensatz zum sequentiellen Programm wird in der nebenläufigen Variante ein weiterer Parameter an das Programm übergeben. Dieser bestimmt die Anzahl der asynchronen Aktivitäten, die zur Berechnung benutzt werden sollen. Der Start dieser Aktivitäten erfolgt in der äußeren `for`-Schleife mit den angehängten `async`-Statement. Umgeben ist die äußere Schleife zudem noch von einem `finish`-Statement, welches die Berechnung von  $\pi$  erst durchführt, wenn alle Aktivitäten ihr Ergebnis in `result` abgelegt haben. Da sich alle Aktivitäten an der gleiche Place befinden, muss der Zugriff auf `result` synchronisiert werden. Dazu wird ein atomarer Block benutzt. Alternativ hätte man auch Instanzen von `AtomicInteger` oder `AtomicDouble` verwenden können. Diese haben die gleiche Funktionalität wie die gleichnamigen Klassen im Java-Paket `java.util.concurrent`.

```

4 public static def main(args:Array[String]) {
5     val N = Int.parse(args(0));
6     val A = Int.parse(args(1));
7     val result = new Cell[Double](0);
8     finish for (1..A) async {
9         val R = new Random();
10        var myResult:Double = 0;
11        for (1..(N/A)) {
12            val x = R.nextDouble();
13            val y = R.nextDouble();
14            if (x*x + y*y <= 1) myResult++;
15        }
16        atomic result() += myResult;
17    }
18    val pi = 4*(result())/N;
19    Console.OUT.println("The value of pi is " + pi);
20 }

```

Abbildung 10: Die nebenläufige Implementierung von MontyPi

### 4.2.3 verteilt

Die dritte Implementierung in Abbildung 11 erweitert die nebenläufige Variante um eine verteilte Ausführung zu gewährleisten. Die Anzahl der zu startenden Aktivitäten wird dabei nicht mehr durch einen Programmparameter vorgegeben, sondern durch die Anzahl der verfügbaren Places. Die Anzahl der Places ist abhängig von der vorhandenen Umgebung und kann durch den Benutzer vor Programmstart justiert werden. Die äußere `for`-Schleife iteriert nun über Places und startet an jeder vorhandenen Place eine asynchrone Aktivität. Die asynchronen Aktivitäten generieren eine Teilmenge der zufälligen Punkte und zählen die Anzahl der Punkte innerhalb der grünen Fläche. Da sich alle Aktivitäten an unterschiedlichen Places befinden, wird das Ergebnis `result` in einer Variable vom Typ `GlobalRef[Cell[Integer]]` verwaltet. Mithilfe diesen Typs lässt sich die erzeugende Place von `result` an jeder beliebigen Place ermitteln. Dies ermöglicht den temporären Transport der Aktivitäten zur Erzeuger-Place um das individuelle Ergebnis atomar in `result` zu aggregieren. Schlussendlich wird der Kontrollfluss der Hauptaktivität fortgeführt und  $\pi$  berechnet sobald alle Aktivitäten ihre Ausführung abgeschlossen haben.

## 5 Zusammenfassung

X10 bietet eine reichhaltige Plattform für parallele und verteilte Anwendungen verpackt in einer leicht zugänglichen Sprache. Die Plattform abstrahiert von konkreten Implementierungsdetails

```

4 public static def main(args:Array[String]) {
5     val N = Int.parse(args(0));
6     val result = GlobalRef[Cell[Double]](new Cell[Double](0));
7     finish for (p in Place.places()) async at (p) {
8         val R = new Random();
9         var myResult:Double = 0;
10        for (1..(N/Place.MAX_PLACES)) {
11            val x = R.nextDouble();
12            val y = R.nextDouble();
13            if (x*x + y*y <= 1) myResult++;
14        }
15        val ans = myResult;
16        at (result) atomic result()() += ans;
17    }
18    val pi = 4*(result()())/N;
19    Console.OUT.println("The value of pi is " + pi);
20 }

```

Abbildung 11: Die verteilte Implementierung von MontyPi

durch zahlreiche elegante Sprachkonstrukte. Die Hürde für den Einstieg in die Entwicklung mit X10 ist somit deutlich niedriger als mit anderen Frameworks oder Schnittstellen. Aufgrund der hohen Abstraktion kann es allerdings passieren, dass die Performanz an gewissen Stellen nicht mit reinen MPI- oder OpenMP-Programmen mithalten kann. Mögliche Gründe wie der häufige Transport von Aktivitäten über ein Cluster ist aufgrund des Abstraktionslevels nicht so transparent sichtbar wie bei anderen Techniken.

Trotz der guten Verknüpfung von Performanz und Skalierbarkeit bei einer hohen Produktivität, steht die Plattform noch am Anfang. Die Verwendung von X10 ist nach 7 Jahren Entwicklung immer noch eindeutig in der Forschung angesiedelt. Trotz massiv steigender Anzahl an Publikationen mit Bezug zu X10, stößt man in diversen Dokumenten auf ungelöste Probleme. So merkt man speziell im Programmier-Handbuch<sup>8</sup> als auch in der Spezifikation<sup>9</sup>, dass viele Fragen und manches Verhalten ungeklärt bleiben. Leider werden solche semantischen Unklarheiten nicht immer direkt gekennzeichnet. Diese Lücken in der Spezifikation führen allen Anschein nach zur bisherigen Missachtung der X10-Plattform im industriellen Kontext.

<sup>8</sup><http://x10.sourceforge.net/documentation/guide/2.1.0/pguide.pdf>

<sup>9</sup><http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf>

## Literatur

- [CGS<sup>+</sup>05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM, 2005.
- [FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 212–223. ACM, 1998.
- [PZ97] Michael Philippsen and Matthias Zenger. Javaparty: Transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
- [SAB<sup>+</sup>10] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. Technical report, June 2010.