Alexander von Rhein

Seminararbeit
Seminar “Multicore Programmierung”
Leitung: Prof. Christian Lengauer
SS 2009, Universität Passau

Die Programmiersprache X10

Diese Seminararbeit beschäftigt sich im Kontext des Seminars „Multicore Programmierung“ mit der neuen Programmiersprache X10.

X10 wird von den IBM Labs in Kooperation mit akademischen Partnern entwickelt um Hochleistungsrechnen auf parallelen Computersystemen zu unterstützen. Die Sprache soll eine möglichst einfache und verständliche Semantik besitzen und gleichzeitig effiziente Entwicklung von hochperformanten, parallelen Code ermöglichen. Dafür werden möglichst viele verschiedene, inhomogene Hardwarestrukturen unterstützt.

Inhalt

1. Einleitung

X10 ist eine klassenbasierte, objektorientierte Programmiersprache mit Einfachvererbung die für hochperformante, parallele Programme auf Hochleistungscomputersystemen entworfen wird. X10 ermöglicht es mit hoher Produktivität effiziente, parallele Programme zu entwickeln. X10 befindet sich noch in einem relativ frühen Entwicklungsstadium. Sie wird speziell für so genannte NUCC (Non-Uniform-Cluster Computing) Systeme optimiert. Diese Systeme bestehen aus (potentiell verschiedenen) Prozessoren die selbst mehrere Kerne haben können und untereinander verbunden sind.

X10 unterstützt spezielle Sprachkonstrukte die es ermöglichen relativ schnell mächtige parallele Programme zu schreiben. Die Entwickler erhoffen sich damit die Entwicklung von effizienten parallelen Programmen soweit zu vereinfachen das eine breite Masse von Programmierern in der Lage ist möglichst produktiv Software für diese Hochleistungssysteme zu entwickeln. Der Hintergrund für diese Notwendigkeit ist, das man in Zukunft konventionelle Prozessoren nicht wie in der Vergangenheit durch Erhöhung der Taktrate beschleunigen kann. Darum erwartet man das NUCC Systeme in Zukunft für ein breiteres Spektrum von Anwendungsgebieten eingesetzt werden. Die Entwickler sind der Meinung, dass Aspekte der Lokalität und Verteilung von Daten und Berechnungen in hochperformantem, parallelem Code nicht vor dem Programmierer versteckt werden sollte (1). Darum wird das Programm grundsätzlich schwerer zu verstehen sein als ein vergleichbares sequentielles Programm. Mit X10 soll dennoch eine Sprache entwickelt werden, die entsprechende parallele Konstrukte anbietet und gleichzeitig noch möglichst verständlich und produktiv ist.

X10 wird von IBM im Rahmen des IBM PERCS (Productive Easy-To-Use Reliable Computer Systems) Projekts mit akademischen Partnern entwickelt. Die Webseite von X10 ist unter der folgenden URL zu finden: http://x10.codehaus.org.

In dieser Seminararbeit wird die Sprache X10 zusammenfassend dargestellt. Aufgrund von Umfangslimitierungen kann nicht auf jede interessante Einzelheit eingegangen werden. Besonderer Fokus wird auf die Unterschiede zu anderen Programmiersprachen /-Konzepten (zum Beispiel JAVA) und auf Konzepte zur Parallelprogrammierung in X10 gelegt. Außerdem werden die Gründe und Ziele die zu Designentscheidungen in der Entwicklung von X10 geführt haben diskutiert.

Da X10 eine Sprache in der Entwicklung ist werden Sprachkonstrukte immer wieder verändert und aus der Sprache entfernt/ hinzugefügt. Sämtliche in dieser Arbeit dargestellten Code-Beispiele laufen mit dem X10DT-Plugin für Eclipse das die X10 Version 1.5 unterstützt. Die für diese Arbeit verwendete „Report on the Experimental Language X10“ (2) hat jedoch schon die Version 1.73. Weiterhin wurde am 26.6.2009 die Version 1.75 veröffentlicht, die für diese Arbeit nicht mehr berücksichtigt werden konnte.

2. Motivation für die Entwicklung von X10

2.1. Ziele

Um den oben angesprochenen Problemen entgegenzuwirken haben die Entwickler von X10 vier wichtige Ziele formuliert die X10 erfüllen soll (3):

Safety. Die Programmiersprache soll sicher sein, um eine hohe Produktivität zu ermöglichen. Einige Fehler, die in Hochperformanz- Programmen häufig vorkommen, sollen schon durch das Sprachdesign verhindert werden. Zu diesen Fehlern gehören ungültige Zeiger, Typfehler und buffer overflows. Durch die Empfehlung von bestimmten Programmier-Mustern soll der Entwickler u.a. in der Vermeidung von Deadlocks unterstützt werden.

Analyzability. X10 Programme sollen von Software Werkzeugen analysierbar sein. Durch die Ergebnisse dieser Analysen kann der Entwickler effektiver arbeiten und Fehler früher feststellen. So könnte ein Analysewerkzeug zum Beispiel alle parallelen Pfade erkennen und grafisch darstellen. Wenn möglich sollte eine formale Semantik für die Sprache entwickelt werden, so dass Programmteile ausgetauscht werden können und die Funktion des Programmes weiterhin garantiert werden kann. Mit entsprechender Tool-Unterstützung (z.B. Eclipse) wäre es dann kosteneffizient möglich das Programm zu analysieren auf bestimmte Architekturen zu optimieren. Es existieren bereits Arbeiten, die die Analysierbarkeit von X10 ausnutzen um vom Benutzer ausgewählte Teile einer Schleife parallel mit anderen Iterationen der Schleife ausführen zu können (4).

Scalability. Skalierbarkeit bedeutet, dass das Hinzufügen von Ressourcen (Speicher, Prozessoren) zu einer Verbesserung der Ausführung eines Programms führt. Die Skalierbarkeit von Programmen hängt im Wesentlichen nicht von der Programmiersprache sondern von dem verwendeten Algorithmus ab. Eine Programmiersprache kann den Entwickler aber darin unterstützen kritische Punkte in der Programmausführung zu identifizieren, und ihm Alternativen bieten um diese Punkte skalierbar zu gestalten.

Flexibility. Skalierbare Programme auf NUCC Systemen werden mehrere Parallelitätsarten auf unterschiedlichsten Architekturen ausnutzen müssen. Dazu gehören Datenparallelität, Ausführungsparallelität, pipelining und parallele Eingaben an das Programm (3). Um dies zu ermöglichen müssen das Datenmodell, das Parallelitätsmodell und der Distributionsmechanismus möglichst allgemein und flexibel gestaltet sein.

2.2. Design Entscheidungen

Aufgrund dieser Ziele wurden einige grundsätzliche Entscheidungen getroffen:

Neue Programmiersprache. Anstatt Sprachbibliotheken (z.B. MPI) oder annotationsbasierte Mechanismen für existierende Sprachen zu entwickeln haben sich die Designer von X10 entschieden eine neue Sprache zu entwickeln, die auf existierenden, etablierten, objektorientierten Sprachen aufbaut.

OOP basiert. Als Ausgangspunkt für die Entwicklung von X10 wird die Programmiersprache JAVA dienen. Diese Sprache ist weitverbreitet und bringt ein sehr großes System an Tools und Dokumentationen mit sich. X10 wird ein *extended Subset* von JAVA sein. Das bedeutet, dass viele

Sprachelemente von JAVA übernommen werden um Programmierern das Erlernen der neuen Sprache zu erleichtern. Einige Sprachelemente werden nicht übernommen und einige neue Elemente werden eingebracht. X10 unterscheidet, wie JAVA, zwischen *value*- Typen (immutable) und *reference*- Typen. Die primitiven Typen aus JAVA (int, double, boolean, ...) werden als *value* Typen implementiert.

Außerdem wird das Konzept der JAVA Virtual Machines (VMs) übernommen. Der X10- Compiler wird, ähnlich wie in JAVA Bytecode produzieren, der von X10- VMs interpretiert und ausgeführt wird.

Auf die einzelnen Unterschiede zu JAVA wird im Kapitel 3 eingegangen.

Partitioned Global Address Space. X10 führt einen *partitioned global address Space (PGAS)* ein, wobei jede Lokalität als *Place* referenziert wird. Ein PGAS ist ein globaler Adressbereich, das bedeutet das jedes Objekt global gesehen eindeutig referenziert werden kann. Dieser globale Adressbereich wird partitioniert indem jedes Objekt genau einer Lokalität (Place) zugeordnet wird. Dem Entwickler wird so die Möglichkeit gegeben zu bestimmen an welcher Lokalität welche Daten gespeichert werden. Dadurch kann er die Vorteile von lokalem Speicher ausnutzen. JAVA Systeme haben nur einen einzigen Heap und es ist schwierig diesen auf verteilte Speichersysteme aufzuteilen. Dieses Problem wird durch das PGAS Konzept gelöst.

Dynamische, Asynchrone Aktivitäten. Die Entwickler von X10 sind der Meinung das die Mechanismen die JAVA für intra-node Parallelismus (threads) und inter-node Parallelismus (messages, processes) zur Verfügung stellt zu schwergewichtig und umständlich sind um damit große, parallele Systeme zu programmieren. In X10 wurden leichtgewichtige, asynchrone *Activities* eingeführt um Threads zu ersetzen. Threads können auch zur Kommunikation zwischen Kernen (bzw. *Places*) genutzt werden.

Array Sub-Language. X10 enthält viele Mechanismen um Arrays auf Lokalitäten zu verteilen, und parallel zu verarbeiten. Weil Arrays besonders im *High Performance Computing (HPC)* eine wichtige Rolle spielen wurde besonders auf diese Array Mechanismen Wert gelegt. Sie werden im Einzelnen im Kapitel 4 beschrieben.

3. Unterschiede zu Java

Wie oben schon erwähnt sind die sequentiellen Teile der Sprachen JAVA und X10 sehr ähnlich. In X10 werden auch Klassen und Interfaces geschrieben. Klassen können höchstens eine Superklasse angeben (Einfachvererbung) wie in JAVA.

Einige Konzepte von JAVA werden ausdrücklich nicht in die Sprache übernommen. Hier sind vor allem die Threads zu nennen. Sie werden durch asynchrone *activities* ersetzt. *Locks* auf Objekten werden durch *atomic Sections* ersetzt. Die Behandlung von Exceptions wird an die geänderte Semantik angepasst.

Barriersynchronisation, wie sie zum Beispiel in MPI durchgeführt wird, werden durch *Clocks* ersetzt. Außerdem enthält in X10 weitere Konzepte wie (*places, regions* und *distributions*). Die X10-spezifischen Konstrukte werden im Folgenden erläutert.

4. Zusätzliche Konzepte in X10

4.1. Übersicht

In diesem Kapitel werden spezielle X10- Konstrukte behandelt. Weil sich diese Konstrukte auf einander beziehen möchte ich wichtige Begriffe im Voraus festlegen.

Place	Eine lokale (Rechen-) Einheit im NUCC System. Entspricht in etwa einer „locale“ in Chapel (3).
Activity	Logische Ausführungseinheit. Entspricht einem Thread in JAVA. Bei X10 wird jedoch viel Wert auf Leichtgewichtigkeit von Activities gelegt.
Array	In X10 können Arrays auf mehrere Places verteilt werden. Dafür gibt es spezielle Sprachkonstrukte (Distributions).
Distribution	Ordnet jedem Element einer Menge von *Points* (z.B. Array Elementen) einen Place zu.
Clock	Werden zur Synchronisation von Activities genutzt. Clocks ersetzen Barriers, wie sie zum Beispiel in MPI verwendet werden.
Atomic Block	Ausführungseinheit bei deren Ausführung garantiert wird, dass sie exklusiven Zugriff auf Daten hat. Es gibt *conditional* und *unconditional* Atomic Blocks.
Reference Classes	Veränderbar, bleiben immer auf einem Place
Value Classes	Unveränderbar, können frei zwischen Places kopiert werden

4.2. Places

Ein *place* entspricht einer Ausführungeinheit des Multicore Systems auf dem X10 ausgeführt wird. Daten und *Activities* sind jeweils einem Place zugeordnet. *Activities* können effizient auf Daten zugreifen, falls diese auf demselben *Place* liegen.

Ein *place* ist konzeptionell ein „virtueller shared-memory multi-Prozessor“ (2). Er hat eine veränderbare, aber nach oben beschränkte Anzahl von Hardware-Threads, die auf einen gemeinsamen, beschränkten Speicherbereich zugreifen.

Places können mit Hilfe der Klasse `x10.lang.place` wie normale Objekte im Programmcode behandelt werden. Die Anzahl der *Places* wird zum Anfang der Ausführung des Programmes festgelegt und bleibt konstant. Mit dieser Design- Entscheidung bleiben die X10-Entwickler konform zu existierenden Modellen wie MPI und OPENMP, bei denen am die Anzahl der Rechenknoten zu Beginn der Ausführung feststeht. Die Entwickler behalten sich jedoch vor in späteren Versionen der Sprache die Erzeugung von neuen *Places* zuzulassen (3). Diese würden dann einem bisher ungenutzten Knoten zugewiesen, oder virtuell realisiert.

Die Zuordnung von Places zu physikalisch vorhandenen Recheneinheiten im NUCC System wird

von einer *Deployment*- Schicht vorgenommen. Diese ist unabhängig vom eigentlichen X10- Programm.

In einer *Activity* kann der Place auf dem sie ausgeführt wird mittels der Konstanten `here` abgefragt werden. Places werden genutzt um *Activities* auf die parallelen Kerne des Systems zu verteilen. Es existieren spezielle Konstrukte um Arrayelementen bzw. Operationen auf Arrayelementen *Places* zuzuweisen (Kapitel zu *Distributions*, Kapitel 4.4).

4.3. Asynchronous Activities

Ein X10 Programm kann gleichzeitig mehrere *Activities* auf unterschiedlichen *Places* ausführen. Das Activity-Konstrukt ist ähnlich zu Thread-Konstrukten in anderen Sprachen wie JAVA. In X10 ist es aber viel einfacher Activities zu starten. In JAVA muss man den Thread in eine separate `Runnable`-Klasse auslagern und diese mit der Methode `run()` starten. In X10 genügt es zum Beispiel den Befehl `async (P) (S)` auszuführen, wobei `P` ein *Place*, und `S` ein *statement* ist. Außerdem kann man mit einem Befehl gleich mehrere Activities mit einem Befehl starten. Es ist möglich mit einem verteilten Array `A`, für jedes Array-Element `A[i]` auf dem Place dem `A[i]` zugeordnet ist eine *Activity* zu starten. Auf diese Befehle wird im Kapitel zu verteilten Arrays eingegangen (Kapitel 4.6).

Um Activities zu verwalten stehen die Folgenden Befehle zur Verfügung:

- Async (P) (S) Generiert eine neue asynchrone *Activity* und führt sie (irgendwann) aus
- Finish (A) Wartet auf die Ausführung eines Statements S (und aller davon gestarteten Activities)
- Future (P) (E) Ausdruck E wird auf Place P ausgeführt (analog zu async)
- Force (F) Führt ein Future F aus (blockierend)
- Array-/ *Distribution*- abhängige Befehle (werden im Array-Kapitel behandelt, Kapitel 4.6)

Mit den Befehlen Async und Finish kann man nun parallele Activities starten und beenden. In X10 ist es möglich, das eine Activity beendet wird, bevor alle von ihr gestarteten Activities beendet sind. Diese Möglichkeit wird benötigt um die globale Anzahl an Activities möglichst klein zu halten. Die Entwickler von X10 erwarten Programme mit extrem vielen gleichzeitig ausgeführten Activities und entwerfen diese deshalb besonders „leichgewichtig“ und dynamisch (schnelles Starten/ Beenden) (2). Dieses Modell (Activities können beenden bevor ihre „Söhne“ beendet sind) führt zu Problemen bei der *Exception handling* Strategie. Dies wird im Kapitel zu Exceptions (Kapitel 4.8) behandelt.

Da *Activities* nur auf einem Place laufen, und auch nur auf Daten dieses Places zugreifen dürfen, muss eine neue *Activity* gestartet werden, um auf Daten von einem entfernten Place zuzugreifen. Dies wird am einfachsten durch eine `Force Future` Kombination erreicht. Da Zugriffe auf entfernte Speicherplätze relativ kostenintensiv sind möchten die X10-Designer explizit auf diese Kosten aufmerksam machen indem der Programmierer gezwungen wird dieses Konstrukt einzusetzen (es wäre auch möglich das Konstrukt implizit zu generieren) (1).

4.4. Distributions

Distributionen ordnen jedem Element einer Menge von *Points* (`x10.lang.point`) einen Place zu. Diese Zuordnung wird primär genutzt um die Verteilung von Array Elementen auf Places zu steuern. Sie kann jedoch auch genutzt werden um auf bestimmten (oder auf allen) Places *Activities* zu starten.

```
final x10.lang.dist myDist = dist.factory.unique();
finish async {
      // ateach: at remote Places
      ateach (point i:myDist) {
            System.out.println("Place " + here.id + " " + i);
      }
}
```

Dieser Code erzeugt zuerst eine Distribution, die gleichverteilt auf allen Places ist (jeder Place wird einem Point zugeordnet). Dann wird für jeden Point (auf dem jeweiligen Place) eine Activity gestartet. Es gibt verschiedene factory- Methoden um Distributions zu erzeugen (Block-Distribution, zyklische Distribution). Ausserdem kann man durch Operationen (Restriktionen, Vereinigungen, Schnitte, …) diese vorgegebenen Distributionen individuell anpassen. Dadurch erhält man ein sehr mächtiges und flexibles Werkzeug zur Verteilung von Arrays. Für den Zugriff auf Arrays sind die zugrundeliegenden Distributionen aber weitgehend transparent.

4.5. Datenmodell

Klassen können in X10 entweder als Value- oder als Reference Class deklariert werden. Reference Classes entsprechen normalen JAVA Klassen und können beliebig verändert werden. Sie können jedoch nicht zwischen Places verschoben werden. Änderungsoperationen können also auch nur von Activities durchgeführt werden die auf demselben Place laufen, auf dem die Instanz der Reference Class liegt.

Value Classes (und alle ihre Subklassen) sind per Definition unveränderbar (immutable) und zustandslos. Sie können nach ihrer Erzeugung nicht mehr verändert werden und dürfen keine Reference Classes als Subklassen haben. Sie dürfen jedoch Felder haben, deren Typen Reference Classes sind. Diese enthalten eine Referenz rA auf eine Instanz A der Reference Class. Diese Instanz A ist also veränderbar. Die Referenz rA selbst darf nicht verändert werden.

Reference und Value Classes gehören in X10 zu den skalaren Datenobjekten. Sie enthalten eine begrenzte, statisch festgelegte Anzahl von Feldern. Außerdem stellt X10 aggregierte Datenobjekte zur Verfügung. Diese werden durch Arrays implementiert, die im nächsten Kapitel vorgestellt werden.

Weil X10 das PGAS Modell implementiert können alle Objekte von jedem Place aus referenziert werden. Dies könnte naiv über ein zentrales Objekt-Register gelöst werden was jedoch extrem ineffizient wäre. In der konkreten Implementierung werden Objekte über so genannte Fat- Pointer identifi-

ziert. Dieses Konzept wird im Kapitel „Implementierungen von X10“ (Kapitel 6) angesprochen.

4.6. Verteilte Arrays

Ein Array besteht aus einer Menge von *Points* (*Region*) und einer *Distribution*. Die *Distribution* ordnet jedem *Point* des Arrays einen *Place* zu. Ein Array kann aus einer existierenden *Distribution* zum Beispiel mit folgendem Code erzeugt werden.

```
dist myDist = dist.factory.unique();
final int[.] a = new int[myDist] (point p) {return here.id;};

foreach (point p : a) {
      System.out.println(a[p]);
}
```

Dieses Beispiel erzeugt einen integer-Array, der gleichmäßig auf alle Places verteilt ist. Bei der Erzeugung wird in jedes Array Feld von einer Activity auf dem jeweiligen Place die Nummer des Places eingetragen. Diese Nummern werden nachher von der Root-Activity (Activity in der das Hauptprogramm läuft) ausgelesen.

Es gibt zwei Befehle zum Starten von Activities in Bezug auf Arrays und Distributionen:

`foreach` Startet für jedes Element des Arrays (der Distribution) eine Activity auf dem lokalen Place

`ateach` Startet für jedes Element des Arrays (der Distribution) eine Activity auf dem Place auf dem das Element liegt.

Mit diesen Befehlen wird sehr deutlich, wie einfach es ist mit X10 Activities zu erzeugen. Das wird vermutlich dazu führen das viele sehr kleine Activities erzeugt werden. Ein gutes Beispiel hierfür ist die Activity im obigen Codebeispiel, die das Array initialisiert. Dieser Befehl wird auf jedem Place als Activity ausgeführt. Darum muss die Implementierung von Activities in X10 extrem „leichgewichtig“ und effizient sein.

4.7. Clocks

Clocks sind zusammen mit atomic Sections die primären Mechanismen in X10 um Activities zu koordinieren (1). In jedem Zeitpunkt der Programmausführung ist eine Clock in X10 in einer bestimmten Phase und hat eine Menge von registrierten Activities. Diese können anzeigen, dass sie für die nächste Phase bereit sind. Wenn alle registrierten Activities bereit sind kann die Clock in die nächste Phase übergehen, und alle registrierten Activities können ihre Berechnung fortsetzen.

In anderen Sprachen bzw. Libraries wie zum Beispiel MPI werden statt Clocks Barriers verwendet. In MPI koordiniert ein Barrier alle Prozesse in einem Communicator. Dieser Ansatz hat 2 Probleme:

1. Synchronisation von sich dynamisch verändernden Prozessgruppen ist schwierig zu realisie-

ren.

2. Wenn verschiedene Prozesse auf verschiedene Barriers in unterschiedlicher Reihenfolge warten können Deadlocks entstehen.

(3)

X10 Clocks werden entwickelt um deadlock-freie parallele Ausführung zu garantieren. Der Beweis wird im Paper (5) geführt. Die Idee ist, das Activities nur feststecken können, wenn sie next() auf einer Clock aufgerufen haben. Wenn ein Programm in einem Deadlock ist müssen also alle Activities `next()` auf einer Clock aufgerufen haben. Der `next()` Befehl schaltet aber alle Clocks auf denen die Activity registriert ist in den nächsten Zustand. Also können alle blockierten Activities weiterrechnen und das Programm hat keinen Deadlock.

```
finish async {
   final clock c = clock.factory.clock();
   for (int i = 0; i < 2; i++) {
      async (place.factory.place(i)) clocked (c) {c.next();
         System.out.println("Place " +here.id + " done");
      }
   }
   for (int i = 2; i < 4; i++) {
      async (place.factory.place(i)) clocked (c) {c.resume();
          System.out.println("Place " +here.id + " done");
      }
   }
}
```

In diesem Beispiel für die Verwendung von Clocks werden (auf mindestens 4 Places) Activities erzeugt. Die Activities auf Places 0 und 1 warten blockierend (mit `c.next()`) auf die Clock c während die Activities auf Places 2 und 3 die Clock weiterschalten (mit `c.resume()`) und sofort den Ausgabebefehl ausführen. Bei der Ausgabe werden (meistens) die Ausgaben von Place 2 und 3 vor den Ausgaben von Place 0 und 1 stehen.

4.8. Exceptions

Aufgrund des Activity Konzeptes von X10 ist es möglich, das eine Activity A, die eine Activity B gestartet hat, beendet wird, bevor B beendet wird. Selbst wenn A noch laufen würde könnte eine Exception, die in B generiert wird nicht von A behandelt werden, weil A inzwischen andere parallele Berechnungen durchführt.

Um trotzdem sicheres Exception Handling zu implementieren wurde das *X10 rooted Exception Model* entwickelt. Als Wurzel einer Activity B wird die Activity A definiert, die der nächste Vorfahr von B ist und die betreffende Kind-Activity mit einem blockierenden Befehl (`finish`) gestartet hat. Diese Wurzel-Activity existiert in jedem Fall weil jeder X10 Code von der Laufzeitumgebung mit `finish` gestartet wird. Exceptions die in B auftreten werden an seine Wurzel A weitergeleitet und

dort mit try-catch-Statements behandelt. Es werden also keine Exceptions „vergessen“ wie es zum Beispiel beim JAVA- Threads möglich ist (2).

4.9. Atomic Blocks

Die einfachste Variante von Atomic Blocks in X10 sind *unconditional Atomic Blocks*. Sie werden mit dem Schlüsselwort `atomic` eingeleitet und garantieren, dass der Code im Block exklusiven Zugriff auf verwendeten gemeinsamen Speicher hat. Der exakte Algorithmus wurde in der aktuellen Sprachspezifikation (2) noch nicht angegeben.

In *unconditional atomic Blocks* dürfen keine asynchronen Aufrufe gestartet werden, keine potentiell blockierenden Befehle ausgeführt werden. Damit kann in den Blocks natürlich auch nicht auf den Speicher anderer Places zugegriffen werden.

Atomizität wird nur garantiert wenn der Block keinen Fehler produziert. Falls eine Exception geworfen wird ist der Programmierer für das Zurücksetzen von eventuell veränderten Variablen verantwortlich (2).

Die zweite Variante der Atomic Blocks sind *conditional atomic Blocks*. Diese bestehen aus einer oder mehreren Kombinationen von *Guards* (boolsche Ausdrücke) und *Atomic Blocks*. Wenn einer der *Guard*-Ausdrücke zu true auswertet wird der dazugehörige *Atomic Block* ausgeführt. Die Activity in der der *unconditional Atomic Block* auftritt blockiert bis einer der Blöcke ausgeführt wurde.

Mit diesem Konstrukt können Deadlocks konstruiert werden. Es wurde wegen seiner Mächtigkeit in die Sprache aufgenommen, die X10 Entwickler empfehlen aber es möglichst nicht zu benutzen weil sich die meisten Probleme auch mit schwächeren, sicheren Mitteln lösen lassen (3).

5. Safety

X10 gibt dem Programmierer einige Sicherheitsgarantien die zu großen Teilen von verbreiteten Sprachen wie JAVA und C# übernommen wurden. In verschiedenen Quellen werden unter anderem Typsicherheit, Speichersicherheit, Place-Sicherheit und Clock-Sicherheit (Deadlock-Freiheit) genannt [z.B. in (6)]. Sicherheit muss man in diesem Zusammenhang als eine eingeschränkte Sicherheit verstehen. Es ist möglich die Sicherheiten zu umgehen und Fehler herzuleiten aber man muss es mehr oder weniger absichtlich machen. Ein gutes Beispiel hierfür ist die Deadlock-Freiheit. X10 garantiert Deadlock-Freiheit solange man keine *conditional atomic Blocks* verwendet. Sobald man jedoch für diese Konstrukte *Guards* (Kapitel über Atomic Blocks, Kapitel 4.9) definiert, die nie wahr werden hat man einen Deadlock.

6. Implementierungen von X10

Es existiert eine Implementierung eines großen Teils der X10 Spezifikation die jedoch bisher nur auf Single Processor Systemen läuft. Diese *Referenz-Implementierung* wird realisiert indem man den X10- Quellcode entweder in JAVA- Code übersetzt und ihn mit speziellen Bibliotheken auf einer JAVA- Umgebung ausführt oder ihn in C (++) Code übersetzt. Für die Entwicklung in der JAVA Umgebung existiert ein Eclipse-Plugin das jedoch bisher nicht die neueste Sprach-Spezifikation unterstützt. Diese Implementierung ist nicht für den Produktiv-Einsatz gedacht, weil sie nicht effizient auf einem NUCC-System laufen kann. Sie wird nur zur Entwicklung der Sprache genutzt.

Weiterhin wird eine *Scalable Optimized Implementation* entwickelt, die auf der High-Performance communication Library LAPI von IBM aufbauen soll. Um dies zu realisieren müssen verschiedene Design Probleme gelöst werden. Einige dieser Probleme werden im Folgenden kurz angesprochen.

Die Memory Management und Garbage Collection Mechanismen müssen erweitert werden um Objektreferenzen die über Clustergrenzen hinausgehen zu unterstützen. Weil X10 Programme sehr viele Activities erzeugen, ausführen und schließen werden muss ein effizientes, leichtes Threading-Modell entwickelt werden. Es könnte zum Beispiel Threads/ Activities die gerade blockiert sind für andere Aufgaben verwenden (dafür müsste der aktuelle Zustand gesichert werden). Ausserdem soll der Scheduler so implementiert werden, dass er Blockierung von Activities möglichst vermeidet. Für die Ermittlung welche Activites parallel ausgeführt werden können ohne deadlocks zu riskieren wurde die May-Happen-in-Parallel (MHP) Analyse (7) entwickelt.

Um das PGAS- Adresskonzept zu effizient verwirklichen werden *Fat-Pointers* verwendet. Weil jede Activity jedes Objekt auf jedem Place adressieren können soll müssen eindeutige Pointer existieren. Das könnte mit einem zentralen Register ermöglicht werden, was jedoch ineffizient wäre. Stattdessen werden Objekte global über *Fat-Pointer* adressiert. Diese enthalten die ID des Places auf dem sie erschaffen wurden (und damit liegen) und eine Place interne ID.

In zukünftigen Implementierungen von X10 sollen auch moderne Sprachmittel wie Generics aufgenommen werden, die man bisher nicht berücksichtigt hat um sich auf die wesentlichen Mechanismen zu konzentrieren.

7. Conclusion

X10 ist eine experimentelle Programmiersprache, die versucht dem Programmierer eine einfache Möglichkeit zu bieten komplexe parallele Programme zu formulieren. Die Entwickler erwarten, das so genannte NUCC (Non-Uniform Cluster Computing) Systeme in Zukunft auch außerhalb von Hoch-

leistungsrechenzentren genutzt werden. Um diese Systeme auszunutzen sollen mehr Entwickler die Möglichkeit erhalten effiziente, objektorientierte, parallele Programme zu schreiben. Ein Ziel bei der Entwicklung von X10 ist es also dem Programmierer die ganze Effizienz des verwendeten Hardware Systems zu bieten und gleichzeitig die Programmiersprache möglichst einfach zu gestalten.

Da die Sprache noch in der Entwicklung ist existiert bisher nur die Spezifikation der Ausdrucksmöglichkeiten, die dem Programmierer geboten werden um parallele Algorithmen zu formulieren. Diese Spezifikation wird ständig weiterentwickelt und ändert sich. Die für diese Arbeit verwendete Spezifikation ist „Report on the Experimental Language X10 Version 1.7“ (2).

Die wesentlichen Faktoren für die Bewertung einer parallelen Programmiersprache lassen sich in folgende Kategorien teilen (8):

- Leistung: Erreichen relevante Programme in der Sprache genügend schnelle Laufzeiten?
- Anwendbarkeit: Existieren genügend Systeme auf denen die Sprache (effizient) läuft?
- Benutzbarkeit: Wie gut unterstützt die Sprache Anwendungsdesign, Entwicklung, Testen und Debuggen?

Für X10 kann man bisher praktisch nur den letzten Punkt (teilweise) bewerten. Die anderen Kriterien lassen sich nur theoretisch abschätzen. Ich denke das die Komplexität, die man bewältigen muss um parallele Algorithmen zu implementieren von der Sprache wirklich merklich reduziert wird. Der Programmierer muss gewisse Entscheidungen treffen (z.B. wo ein bestimmtes Objekt gespeichert wird). Diese kann und darf ihm die Programmiersprache nicht abnehmen. Sie kann jedoch die Syntax möglichst einfach und verständlich machen. Dadurch wird nicht nur die Produktivität des einzelnen Softwareentwicklers erhöht, sondern die Integration von X10 in moderne Softwareentwicklungsverfahren wird vereinfacht. So ist es zum Beispiel weit weniger aufwendig ein Programm das in einer Sprache wie X10 geschrieben ist zu verstehen und zu debuggen, als ein Programm das in MPI formuliert wurde. Ich sehe die Gefahr, das sich die Entwickler von X10 verleiten lassen weitere, mächtigere Konstrukte in die Sprache aufzunehmen, oder alternative Formulierungsmöglichkeiten anzubieten. Damit würde man die Sprache schnell viel komplizierter machen und viele Vorteile die sie bisher über z.B. MPI hat aufgeben.

8. Literaturverzeichnis

1. *X10: Programming for hierarchical parallelism and nonuniform data access (extended abstract).* **Ebcioglu, Kemal, Saraswat, Vijay und Sarkar, Vivek.** 2004. Language Runtimes '04 Workshop: Impact of Next Generation Processor Architectures On Virtual Machines (colocated with OOPSLA 2004).

2. **Saraswat, Vijay und Nystrom, Nathaniel.** *Report on the Experimental Language X10 Version 1.7.* 2008.

3. *X10: an object-oriented approach to non-uniform cluster computing.* **Charles, Philippe, et al.** s.l. : ACM, 2005. OOPSLA. S. 519-538.

4. **Markstrum, Shane A., Fuhrer, Robert M. und Millstein, Todd D.** Towards concurrency refactoring for x10. *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2009, S. 303-304.

5. **Saraswat, Vijay und Jagadeesan, Radha.** Concurrent clustered programming. *CONCUR 2005 - Concurrency Theory.* 2005, S. 353-367.

6. Web Tutorial. *http://x10.codehaus.org/.* [Online] [Zitat vom: 22. Mai 2009.] http://dist.codehaus.org/x10/tutorial/web-tutorial.pdf.

7. **Agarwal, Shivali, et al.** May-happen-in-parallel analysis of X10 programs. *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2007, S. 183-193.

8. **Szafron, Duane und Schaeffer, Jonathan.** An Experiment to Measure the Usability of Parallel Programming Systems. *1996.*