Seminar: Sprachen für Parallelprogrammierung

Chapel

KIT - Universität des Landes Baden-Württemberg und
nationales Großforschungszentrum in der
Helmholtz-Gemeinschaft

IPD Snelting, Lehrstuhl Programmierparadigmen

Dennis Appelt

19. Juli 2010

Inhalt

1. Einleitung 3
2. Anforderungen an ein modernes Sprachdesign 4
   2.1. Aufgaben des Programmierers 4
   2.2. Anforderungen an die Sprache und den Compiler 4
3. Die Sprache Chapel 7
   3.1. Domain und Array 7
   3.2. Daten- und Task-Parallelismus 8
3.3. Datenlokalität 9
4. Zusammenfassung und Vergleich 10
Literaturverzeichnis 11

1. Einleitung

Unter Verwendung der heute üblichen Sprachen zur Parallelverarbeitung ist es ein langer Weg vom Identifizieren von parallel ausführbaren Aufgaben bis hin zur konkreten Implementierung von parallelem Code. Die Ursache hierfür liegt hauptsächlich in einer sehr niedrigen Sprachabstraktionsebene für Parallelismus, wodurch der Programmierer beim Verfassen von parallelem Code gezwungen ist jedes Detail zu berücksichtigen. Als Folge sinkt die Produktivität dramatisch.

Eine weitere Auswirkung dieser low-level Sprachabstraktion ist neben der niedrigen Produktivität die geringe Programmierbarkeit. Nur ein Bruchteil der Gesamtheit aller Programmierer, die in der Lage sind, sequentielle Programme zu schreiben, können äquivalente und fehlerfreie parallele Programme schreiben. Durch die hohe Komplexität des resultierenden Codes, die bei Verwendung von low-level Sprachkonstrukten zwangsweise entsteht, geht eine erhöhte Fehlerwahrscheinlichkeit mit einher. Ein Zitat von einem unbekannten Author beschreibt diesen Sachverhalt treffend: „From ten programmers, elven are unable to write multithreaded code. “

In dieser Arbeit wird eine neue Sprache zur Parallelverarbeitung namens Chapel vorgestellt. Insbesondere sollen die Mechanismen zur Steigerung der Produktivität und der Programmierbarkeit betrachtet werden. Abschließend wird ein Vergleich mit den zu Chapel konkurrierenden Sprachen Fortress und X10 hergestellt.

In Kapitel 2 „Anforderungen an ein modernes Sprachdesign“ werden aus den Mängeln heutiger Sprachen für die Parallelverarbeitung Anforderungen für das Design zukünftiger Sprachen abgeleitet. In Kapitel 3 „Die Sprache Chapel“ wird die Umsetzung der Anforderungen in der Sprache Chapel beleuchtet. Abschließend werden in Kapitel 4 „Zusammenfassung und Vergleich“ die wichtigsten Mechanismen von Chapel wiederholt und ein Vergleich mit konkurrierenden Sprachen für Parallelverarbeitung angestellt.

2. Anforderungen an ein modernes Sprachdesign

Bevor die Anforderungen an den Compiler formuliert werden, soll zuerst eine grundlegende Aufgabentrennung zwischen Programmierer und Compiler vorgenommen werden.

2.1. Aufgaben des Programmierers

Die Autoren der Sprache Chapel sind der Ansicht, dass Techniken, die ohne Zutun des Programmierers automatisch aus sequentiellem Code parallelen Code erzeugen, nicht für den Einsatz in der Sprache Chapel geeignet sind (Parallel Programmability and the Chapel Language, S. 2). Aus diesem Grund wird in Chapel nicht der Weg gegangen, dem Programmierer die komplette Arbeit des Parallelisierens abzunehmen, sondern es sollen bessere Mechanismen und Sprachkonstrukte zur Verfügung gestellt werden um Parallelität auszudrücken. Demzufolge sind die Aufgaben des Programmierers:

Parallelität identifizieren. Potenziell parallel ausführbare Codesegmente müssen vom Programmierer identifiziert werden. Darüber hinaus soll es nicht zwangsläufig zu den Aufgaben des Programmierers gehören, die tatsächliche Umsetzung, in z. B. Threads oder Interprozesskommunikation, zu implementieren.

Synchronisation. Der Bedarf an Synchronisation muss weiterhin vom Programmierer erkannt und auch ausgedrückt werden. Auch hier sollen high-level Abstraktionen, die zum Beispiel mächtiger sind, als die low-level Konzepte des Mutex oder des Lock, zur Verfügung gestellt werden.

Datenverteilung und Lokalität. Zumindest Performance-orientierte Programmierer müssen die Aufgabe der Datenverteilung und Lokalität wahrnehmen.

Da bei allen genannten Aufgaben die Verantwortung in letzter Instanz bei dem Programmierer verbleibt, dieser aber so gut wie möglich durch Sprachabstraktionen und Mechanismen unterstützt werden soll, kann man das Leitmotto der Autoren von Chapel als „specifying intend rather than mechansim“ zusammenfassen.

2.2. Anforderungen an die Sprache und den Compiler

Globale Sicht auf ein Programm

Die Anforderung, dass ein Programmiermodell eine globale Sicht auf ein Programm bieten muss, ist nach Ansicht von [Wei07 S. 4] die wichtigste Anforderung. Um die Vorteile einer globalen Sicht besser nachvollziehen zu können, sollen zuerst die Implikationen durch eine fragmentierte Sicht betrachtet werden.

Bei einer fragmentierten Sicht wird ein Programm mit der Bewusstheit im Hinterkopf geschrieben, dass mehrere Instanzen des Programms gleichzeitig das Gesamtproblem bearbeiten und dass jede einzelne Instanz nur eine Teilmenge des Gesamtproblems bearbeitet. Dabei muss der Programmierer für die Aufteilung des Gesamtproblems in Teilprobleme Sorge tragen und für die Kommunikation zwischen den einzelnen Programminstanzen. Ein in der Praxis häufig vorkommendes Muster ist die Verwendung von MPI in einem Computercluster. Hierbei wird auf jedem Knoten des Clusters eine MPI-Programminstanz ausgeführt. Abbildung 1a zeigt ein Beispielprogramm, das durch eine fragmentierte Sicht erstellt wurde. In Zeile 2 wird zuerst die Größe des Teilproblems in Abhängigkeit der Anzahl von Programminstanzen berechnet. In Zeile 3 bis 5 werden Datenstrukturen zum Speichern des lokalen Teilproblems angelegt und Laufvariablen, die das Teilproblem innerhalb des Gesamtproblems abgrenzen. In Zeile 6 bis 16 findet abhängig von der Position der Programminstanz innerhalb der Topologie die Kommunikation mit anderen Programminstanzen statt. In Zeile 17 und 18 wird letztendlich der eigentliche Algorithmus zur Lösung des Problems ausgeführt.

```
1 var n : int = 1000;
2 var locN : int = n / numTasks;
3 var A, B: [ 0 . . locN +1] float;
4 var myItLo : int = 1;
5 var myItHi : int = locN;
6 if (iHaveLeftNeighbor) then
7         send (left, A(1));
8 else
9         myItLo = 2 ;
10 if (iHaveRightNeighbor) {
11        send(right, A(locN));
12        recv(right, A(locN + 1));
13 } else
14        myItHi = locN−1;
15 if (iHaveLeftNeighbor) then
16        recv(left, A(0));
17 forall i in myItLo .. myItHi do
18        B(i) = (A(i−1) + A(i+1)) / 2 ;
```

Abbildung 1a: Beispiel einer fragmentierten Sicht

```
1 var n: int = 1000;
2 var A, B: [1..n] float;
3 forall i in 2..n−1
4         B(i) = (A(i−1) + A(i+1)) / 2 ;
```

Abbildung 1b: Beispiel einer globalen Sicht

Im Gegensatz hierzu wird bei einer globalen Sicht das pro-Instanz Denken überwunden und die zu programmierende Maschine als eine Einheit betrachtet. Es gibt einen globalen Adressraum, der es ermöglicht Algorithmus und Datenstrukturen nicht aufteilen zu müssen. Abbildung 1b zeigt den Code aus Abbildung 1a aus globaler Sicht. Im Vergleich zu dem Code aus Abbildung 1a fällt auf, dass der Code viel kompakter ist und weniger vom eigentlichen Algorithmus ablenkt. Des Weiteren startet das Programm nur mit einem logischen Thread und

erzeugt bei Bedarf (s. Zeile 3) weitere Parallelität. Die Anforderungen an den Compiler sind bei einer globalen Sicht höher, da dieser die Abbildung von globaler auf fragmentierte Sicht übernehmen muss.

Generalisierte Parallelität

Eine weitere Anforderung an eine Sprache ist, dass verschiedene Ebenen von Parallelität ausdrückbar sein sollen. Nach [Cha07] ist in vielen aktuellen Sprachen nur eine Ebene von Parallelität sauber ausdrückbar. Weitere Ebenen sind entweder gar nicht ausdrückbar oder nur unter Zuhilfenahme anderer Sprachen. Beispielsweise ermöglicht es MPI Parallelität nur auf einer sehr groben Ebene auszudrücken, verschachtelte Parallelität wird mit OpenMP erreicht.

Neben verschiedener Ebenen soll auch sowohl Daten-als auch Taskparallelismus unterstützt werden.

Trennung von Algorithmus und Implementierung der Datenstruktur

Änderungen an der Implementierung einer Datenstruktur sollten keine Änderungen an dem Algorithmus, der auf der Datenstruktur operiert, nach sich ziehen. Beispielsweise soll der Algorithmus für eine Matrix-Vektor Multiplikation nicht davon abhängig sein, ob die Matrix im Speicher als dichtbesetzte- oder als dünnbesetzte Matrix implementiert ist.

Obwohl diese Anforderung auch für sequentielle Programme wünschenswert ist, ist sie für parallele Programme noch verschärft, da Aspekte wie Datenverteilung und Kommunikation eine Trennung von Implementierung der Datenstruktur und Algorithmus noch wichtiger machen.

Performance

Performance-orientierter Code steht in einer gewissen Konkurrenz zu Programmierbarkeit, einem der Hauptziele von Chapel. Während Programmierbarkeit durch high-level Abstraktionen erreicht wird, hat Performance-orientierter Code die Eigenschaft, sehr genau zu beschreiben was ausgeführt werden soll. Die 90/10 Regel besagt, dass 90% des Codes 10% der Gesamtlaufzeit in Anspruch nimmt [Pro]. Diese 90% Prozent sollen so schnell und unkompliziert wie möglich formuliert werden können. Die restlichen 10% des Codes sollen bei Optimierungsbedarf mit weiteren Sprachkonstrukten feiner ausgedrückt werden können.

Eine Anforderung an eine Programmiersprache ist es, dass je nach Anspruch an die Performance unterschiedliche Sprachabstraktionsebenen zur Verfügung stehen. Im Rahmen dieser Ausarbeitung sollen hinsichtlich Performancesteigerung nur Sprachkonstrukte untersucht werden, die in Zusammenhang mit Parallelität und Datenverteilung stehen.

3. Die Sprache Chapel

In diesem Kapitel soll die Umsetzung der im vorherigen Kapitel aufgestellten Anforderungen in der Sprache Chapel untersucht werden. Dies wird erreicht, in dem die wichtigsten Sprachfeatures von Chapel vorgestellt werden und gezeigt wird, wie diese Features die Anforderungen lösen.

Die Sprache Chapel wird von dem Unternehmen Cray im Rahmen des DARPA Projekts High Productivity Computing Systems entwickelt. Neben Chapel werden in diesem Projekt auch noch die Sprachen Fortress und X10 entwickelt. Alle in diesem Kapitel gezeigten Codefragmente sind in Chapel geschrieben.

3.1. Domain und Array

Eines der wichtigsten Konzepte in der Sprache Chapel ist das der Domain. Eine Domain ist eine Menge von Indizes, die Struktur und Größe eines Arrays bestimmt. Nachdem eine Domain definiert wurde, können anhand dieser Domain Arrays erstellt werden, die die Eigenschaften der Domain besitzen.

```
1 var D: domain(2) = [1..m, 1..n];
2 var A: [D] float;
3 A(5,3) = 1;
```

Abbildung 2a: Eine Integer-Domain

```
1 var People: domain(string);
2 var Age: [People] int;
3 People += ”John”;
4 Age(”John”) = 62;
```

Abbildung 2b: Eine unendliche Domain

Abbildung 2a und Abbildung 2b zeigen Beispiele für das Zusammenspiel von Domain und Array. In Abbildung 2a wird eine zweidimensionale Domain der Größe *m* und *n* erstellt und ein Array dieser Domain, welches Gleitkommazahlen aufnimmt. Ebenso wie die Domain besitzt das Array zwei Dimensionen. Abbildung 2b zeigt eine unendliche Domain. Diese besitzt zum Zeitpunkt der Erstellung keine feste Größe, sondern definiert nur den Typ der Indizes als *string*. Im Nachhinein kann eine beliebige Anzahl von Indizes hinzugefügt werden, wie in Zeile 3 demonstriert wird. Obwohl in Zeile 2 das Array erstellt wurde bevor die Domain einen Index enthält kann in Zeile 4 über den Index *John* ein Arrayelement angesprochen werden. Dies zeigt, dass während der kompletten Lebenszeit des Arrays eine lebendige Verbindung zu der Domain besteht. Des Weiteren steht in Chapel der Begriff Array nicht nur für eine Datenstruktur fester Größe auf die durch ganzzahlige Indizes zugegriffen wird, sondern Array bezeichnet eine allgemeine Datenstruktur hinter der sich zum Beispiel ein Set, ein assoziatives Array oder ein Graph verbergen kann. Da Implementierungsdetails, wie zum Beispiel Iterationsmethoden oder der interner Aufbau der Datenstruktur, in der Domain gekapselt sind, muss beim Formulieren eines Algorithmus auf diese Details keine Rücksicht genommen werden. Durch diese „Separation of Concerns“ wird die Anforderung der Trennung von Algorithmus und Implementierung der Datenstruktur möglich.

Des Weiteren besitzt jede Domain eine Distribution. Eine Distribution bestimmt, wie die Indizes einer Domain und somit auch die Arrayelement auf die einzelnen

Knoten eines Clusters verteilt werden. Somit ist der Compiler in der Lage, die globale Sicht auf die fragmentierte Sicht abzubilden. Ein Algorithmus, der auf ein Array zugreift muss nicht mehr explizit ausdrücken, auf welchem Knoten der Zugriff stattfindet. Die Forderung nach einer globalen Sicht auf ein Programm wird somit durch einen, aus Sicht des Algorithmus, globalen Adressraum gelöst.

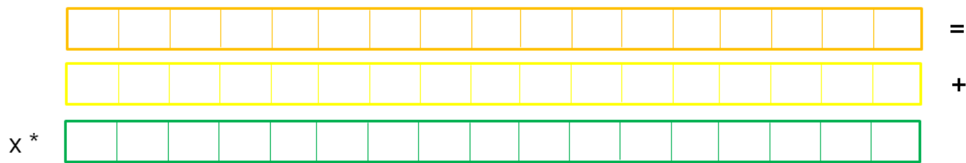

Bild 1: Globale Sicht auf eine Datenstruktur und einen Algorithmus

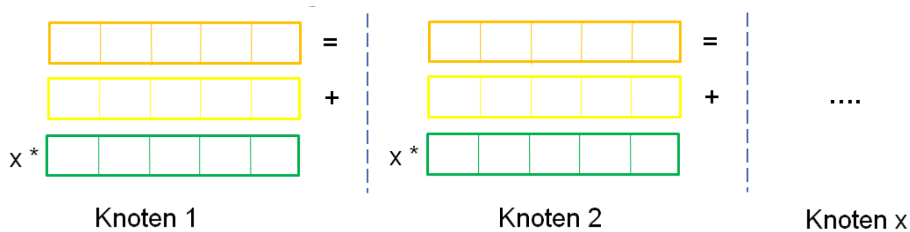

Bild 2: Fragmentierte Sicht auf eine Datenstruktur und Algorithmus

Chapel bietet eine Reihe von Out-of-Box Distributionen wie zum Beispiel eine Block-orientierte oder eine zyklische Distribution. Nichts desto trotz können auch an die eigenen Bedürfnisse angepasste Distributionen erstellt werden.

3.2. Daten- und Task-Parallelismus

Zu Beginn werden Mechanismen gezeigt, um in Chapel Daten-Parallelismus zu erreichen. Die in Abbildung 2a definierte Domain und Array werden in Abbildung 3a und 3b weiterverwendet.

```
1 var innerD: subdomain(D) = [2..m-1,
          2..n-1];
2 forall ij in innerD {
3                  A(ij) = ...
4 }
```

Abbildung 3a: Slicing und paralleler Datenzugriff

```
1 var rnmu = max reduce abs(A);
```

Abbildung 3b: Promotion und Reduce

In Zeile 1 von Abbildung 3a wird das Slicing gezeigt. Beim Slicing wird aus allen Indizes einer Domain eine Teilmenge gebildet. Anschließend wird in Zeile 2 ausschließlich über die in der Teilmenge enthaltenen Indizes iteriert. Durch Verwendung der forall-Schleife wird dem Compiler signalisiert, dass alle Schleifendurchläufe parallel ausgeführt werden können. In Abbildung 3b wird die Promotion und Reduce gezeigt. Bei der Promotion wird einer Funktion, die normalerweise einen skalaren Wert als Parameter nimmt, ein Array übergeben. Dies hat zur Folge, dass die Funktion für jedes Arrayelement aufgerufen wird. Somit wird durch *abs(A)* für jedes Element des Arrays der Absolutbetrag

berechnet. Auf das resultierende Array wird anschließend durch das Schlüsselwort reduce eine Reduktion ausgeführt. Bei einer Reduktion werden jeweils zwei Elemente an die Reduktionsfunktion *max* weitergereicht, welche das Größere der beiden Elemente zurückgibt. Dies wird solang wiederholt bis das größte Arrayelement gefunden ist. Sowohl Promotion als auch Reduce können parallel ausgeführt werden.

Chapel bietet zum Formulieren von Task-Parallelismus die Schlüsselworte begin und cobegin. Durch begin wird die jeweils folgende Anweisung in einem eigenen Ausführungsfaden bearbeitet. Demzufolge wird in Abbildung 4a die Methode *DoThisTask()* in einem separaten Ausführungsfaden bearbeitet. Die Funktion *TheOriginalThread()* in Zeile 2 wird wieder vom Hauptthread ausgeführt. Abbildung 4b zeigt die Verwendung von cobegin. Für jede einzelne Anweisungen, die sich innerhalb des cobegin-Blocks befindet, wird ein eigener Ausführungsfaden gestartet.

```
1 begin DoThisTask();
2 TheOriginalThread();
```

Abbildung 4a: Das Schlüsselwort begin

```
1 var pivot = computePivot(lo, hi, data);
2 cobegin {
3         Quicksort(lo, pivot, data);
4         Quicksort(pivot, hi, data);
5 }
```

Abbildung 4b: Das Schlüsselwort cobegin

Der Anforderungen nach generalisierter Parallelität wird in Chapel entsprochen, in dem sowohl verschiedene Ebenen von Parallelität ausdrückbar sind als auch das Daten- und Task-Parallelismus gleichermaßen unterstützt werden. Konstrukte wie zum Beispiel die forall-Schleife oder das begin-Statement können Verwendung finden, wenn weitere Ebenen von Parallelität nötig sind.

3.3. Datenlokalität

Will ein Programmierer aus Gründen der Performance steuern auf welchem Knoten eine Berechnung stattfinden soll kann er das on-Statement verwenden. Dies kann zum Beispiel von Vorteil sein, wenn der Programmierer einen Wissensvorsprung gegenüber dem Compiler hat und er unter Verwendung des on-Statements die Datenlokalität steigern möchte.

Abbildung 5a und 5b zeigen verschiedene Möglichkeiten das on-Statement zu verwenden. In Abbildung 5a wird die Methode *computeTask()* unabhängig von sonstigen Einflussfaktoren immer auf dem Knoten, der durch *Locales(0)* bezeichnet wird, ausgeführt. *Locales* ist dabei eine Datenstruktur die nativ in jedem Chapel-Programm vorhanden ist und durch die sich die einzelnen Knoten explizit ansprechen lassen. Abbildung 5b nutzt das on-Statement in einer Daten-getriebenen Weise. Die Methode *Quicksort(…)* wird auf dem Knoten ausgeführt, der das Arrayelement *data[lo]* beherbergt.

```
1 on Locales(0) do computeTask(…);
```

Abbildung 5a: on-Statement in Verbindung mit Locales

```
1 on data[lo] do Quicksort(lo, pivot, data);
```

Abbildung 5b: on-Statement in Verbindung mit dem Speicherort

Die Anforderung der Performance lässt sich in Hinblick auf die Datenlokalität durch das on-Statement erreichen. Mit dem on-Statement ist der Programmierer in der Lage bei Bedarf auf einer detaillierteren Ebene eine Aussage über den ausführenden Knoten zu treffen.

4. Zusammenfassung und Vergleich

Alle in Kapitel 2 aufgezählten Anforderungen werden durch Chapel adressiert. Auf die Forderung nach einer globalen Sicht wird mit dem Konzept der Domain und der zugehörigen Distribution reagiert. Auch bei der Trennung von Algorithmus und Implementierung der Datenstruktur ist die Domain das zentrale Konzept. Die Anforderung der generalisierten Parallelität wird unterstützt, indem verschachtelte Ebenen an Parallelität bei Bedarf eingeführt werden können. Außerdem wird Daten- und Task-Parallelität durch zahlreiche Features unterstützt. Zur Steigerung der Datenlokalität und somit der Performance steht das on-Statement zur Verfügung.

Im Vergleich von Chapel zu Fortress und X10, den beiden anderen Sprachen die im Rahmen des Projektes High Productivity Computing Systems entwickelt werden, lässt sich feststellen, dass jede dieser Sprachen die genannten Anforderungen in irgendeiner Form adressiert. Tabelle 1 zeigt die Anforderungen und damit zusammenhängende Features im Vergleich.

Eigenschaften	Chapel	Fortress	X10
Globale Sicht	✓	✓	✓
Generalisierte Parallelität	✓	✓	✓
Knoten explizit ansprechbar	✓	✓	✓
Arrays sind automatisch verteilt	✓	✓	✓
Redistribution von Arrays	✗	✓	✗
Out-of-Box Distributions	✓	✓	✗
Benutzerdefinierte Distributionen	✓	✓	✗

Tabelle 1: Sprachvergleich

Literaturverzeichnis

[Spec] *Chapel Specification v 0.795.* [Online] http://chapel.cray.com/spec/spec-0.795.pdf.

[Ash07] Ashby. 2007. New Languages for High Performance, High Productivity Computing. 2007.

[Cal04] Callahan, Chamberlain and Zima. 2004. The Cascade High Productivity Language. *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments.* 2004.

[Cha07] Chamberlain, Callahan and Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications.* 2007.

[Dia07] Diaconescu and Zima. 2007. An approach to data distributions in Chapel. *International Journal of High Performance Computing.* 2007.

[Pro] Program optimization. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Program_optimization#Bottlenecks.

[Sni06] Snir. 2006. *Programming Languages for HPC: Is There Life After MPI?* [Online] März 2006. http://www.cs.uiuc.edu/homes/snir/PDF/Programming%20Languages%20for%20HPC%20short.pdf.

[Wei07] Weiland. 2007. Chapel, Fortress and X10: novel languages for HPC. 2007.