Universität Passau
Lehrstuhl für Programmierung

Chapel

Seminar: Multicore-Programmierung
Prof. Christian Lengauer

von
Andreas Simbürger
Matrikel-Nr.: 45810
22. Juli 2009

Inhaltsverzeichnis

1 Einführung

In der Einteilung aktueller paralleler Programmiersprachen bzw. Spracherweiterungen existieren derzeit zwei Lager. Auf der einen Seite Konzepte die einem Entwickler eine fragmentierte Sicht auf den zu implementierenden Algorithmus bieten. Dabei wird explizit Kommunikation bzw. Datenorganisation mit dem eigentlichen Algorithmus vermischt und so die Entwicklung unnötig erschwert. Beispiele für eine fragmentierte Sicht sind MPI, OpenMP und pthreads. Die explizit zu führende Kommunikation bzw. Aufteilung der Daten und Synchronisation der Threads machen es sehr schwierig und unkomfortabel effiziente parallele Algorithmen zu entwickeln. Auf der anderen Seite findet man Sprachen, welche ausschließlich eine globale Sicht auf Algorithmen bieten. Die Abstraktion dieser Kategorie ist bereits so stark fortgeschritten, dass man oftmals nicht mehr ausmachen kann an welcher Stelle z.B. Kommunikation stattfindet und an welcher nicht. Man hat kaum Möglichkeiten Detailwissen über einen implementierten Algorithmus zu nutzen und in tiefere Abstraktionsebenen vorzudringen. Die Sprachen ZPL und HPF gehören dieser Kategorie an. Zwischen diesen Kategorien erstreckt sich daher eine große Kluft. Als Entwickler braucht man jedoch eine Sprache, mit der man sowohl auf einer hohen Abstraktionsebene Parallelität implementieren kann, als auch Detailwissen einbringen, indem man explizit Einfluss auf die Parallelität nehmen kann.

Die von Cray im Rahmen des von der DARPA initiierten High Productivity Computer Systems (HPCS) Wettbewerb entwickelte Sprache Chapel versucht mit einem kaskadierenden Ansatz die zwischen den Paradigmen existierende Kluft zu überbrücken. Im folgenden soll eine rudimentäre Einführung in Chapel gegeben werden. Eine ausführliche Beschreibung sämtlicher Konzepte würde den Rahmen einer Seminararbeit bei weitem sprengen. Es wird daher auf die diesen Ausführungen zu Grunde liegende Sprachspezifikation [1] verwiesen.

1.1 Chapel

Chapel, die “Cascade High Productivity Language”, wird von Cray im Rahmen des von der DARPA initiierten High Productivity Computing Systems (HPCS) Wettbewerbs entwickelt. Die Entwicklung an Chapel begann mit dem Start der Phase 2 des Wettbewerbs, parallel zu X10 (entwickelt von IBM) und Fortress (entwickelt von Sun). Seit dem 14. November 2008 steht die Sprache öffentlich unter einer BSD Lizenz zur Verfügung. Nach Beendigung des Wettbewerbs wird eine Übergabe an die Open Source Gemeinschaft angestrebt.

1.2 Sprachfeatures

Chapel bedient sich einiger bereits aus anderen Sprachen bekannten Konzepten und führt diese in einer durch C inspirierten Syntax zusammen. Primär ist Chapel eine blockstrukturierte, imperative Programmiersprache, mit u.a. folgenden Features:

- Verteilte Arrays, Indexmengen
- Taskparallelität, Datenparallelität

- Iteratoren, Generische Programmierung
- Objektorientierte Programmierung
- Automatische Typinferenz (starke Typisierung)

Um einen groben Überblick über die Syntax zu erhalten, sei an dieser Stelle ein "HelloWorld" Programm abgebildet.

```
module Hello {
 config const message = "Hello, world!";

 def main() {
   writeln(message);
 }
}
```

Für weiterführende Informationen bzgl. der allgemeinen Syntax sei hiermit auf die Spezifikation verwiesen.

2 Parallele Sprachkonzepte

Chapel bietet im Gegensatz zu vielen existierenden Sprachen bzw. Spracherweiterungen die Möglichkeit sowohl taskparallel als auch datenparallel vorzugehen. Beiden Ansätzen liegt ein ideales SPMD (Single Process Multiple Data) Modell zu Grunde. Dabei ist zu beachten, dass vollständig von Rechenleistung und lokalen Speichern abstrahiert wird. Im Modell sind alle Knoten (SMP Nodes, CPU Kerne) homogen und der gesamte zur Verfügung stehende Speicher auf Shared-Memory Basis angebunden. Chapel geht immer davon aus, dass Programme stets sequentiell korrekt sind und keine *race conditions* enthalten. Das Unterbinden solcher obliegt der Verantwortung des Entwicklers.

2.1 Taskparallelität

Tasks werden in Chapel stets als Threads implementiert, unabhängig davon ob man sich auf einem Multicore oder auf einem Cluster befindet. Um Taskparallelität zu erzeugen bietet Chapel sowohl strukturierte als auch unstrukturierte Methoden.

2.1.1 Unstrukturierte Taskgenerierung

Die einfachste Methode um in Chapel einen Task zu erzeugen ist das *begin*-Statement.

Syntax

```
begin-stmt:
  begin stmt;
```

Das *begin* Statement erzeugt nicht-blockierend einen parallelen Task und führt *stmt* darin aus. Der Task läuft bis *stmt* terminiert.

Beispiel 1

```
begin writeln("hello world");
writeln("good bye");
```

Dabei werden seitens Chapel keine Maßnahmen bzgl. Speicherschutz getroffen, was zur Folge hat, dass kritische Abschnitte explizit gesperrt werden müssen.

2.1.2 Synchronisation

Um Synchronisation und gegenseitigen Ausschluss zu implementieren bietet Chapel anstatt üblicher Mutex-Objekte einen *sync*-Datentyp.

Syntax

```
sync-type:
  sync type;
```

Jeder primitive Datentyp (z.B. int, bool, real u.a.) kann via Schlüsselwort *snyc* in eine Synchronisationsvariable umgewandelt werden. Dabei erhält die Variable neben ihrem Wert zusätzlich noch zwei Zustände, die zugriffsabhängig angenommen werden. Über Sync-Variablen lässt sich u.a. ein Mutex-Verhalten implementieren.

Standardmäßig folgt eine *sync*-Variable folgendem Schema. Über Zugriffsfunktionen lässt sich jedoch jede mögliche Kombination des Lesens und Schreibens realisieren.

	Lesen	Schreiben
Voll	Leer	Blockieren
Leer	Blockieren	Voll

Beispiel 2

```
var lock: sync bool;

lock = true;
critical();
lock;
```

Prinzipiell hat man damit alle Werkzeuge an der Hand um Multithreaded zu programmieren. Da *begin*-Statements und *sync*-Typen auf einer sehr niedrigen Abstraktionsebene hantieren, benötigt man für eine globalere Sicht andere Syntaxkonstrukte, um strukturiert Taskparallelität zu erzeugen.

2.1.3 Strukturierte Taskgenerierung

Syntax

```
cobegin-stmt:
 cobegin {stmt-list};
```

Das *cobegin*-Statement vereinfacht die Taskgenerierung mit anschließender Synchronisation nach erfolgter Threadtermination. Für jedes Einzelstatement aus der *stmt-list* wird nicht-blockierend genau ein Task erzeugt. Nachdem alle Tasks erzeugt wurden, wird implizit am Ende des Blocks synchronisiert (Impliziter Join). Man beachte, dass ein *cobegin* nur für jedes direkte Element der *stmt-list* eine Synchronisationsvariable erstellt. Auf dynamisch in den Einzelstatements erzeugte Tasks wird nicht gewartet, es bleibt daher dem Entwickler überlassen, innerhalb der Einzeltasks für Synchronisation zu sorgen. Nichtsdestotrotz stellt cobegin nur eine Abstraktion der begin/sync Konstrukte dar und ist daher nicht notwendig.

Beispiel 3

```
cobegin {
 consumer(1);
 consumer(2);
 producer();
}
```

Dieser Block ist äquivalent zu:

```
var s1$, s2$, s3$: sync bool;
begin { consumer(1); s1$ = true; }
begin { consumer(2); s2$ = true; }
begin { producer(); s3$ = true; }
s1$; s2$; s3$;
```

Das *cobegin*-Statement ist hinreichend sofern die Anzahl der zu erzeugenden Tasks gering ist. Sollte man jedoch eine große Menge von Tasks erzeugen wollen bietet sich dafür ein iterativer Ansatz über eine Schleife an.

2.1.4 Iterative Taskgenerierung

Syntax

```
coforall-stmt:
 coforall index-expr in iterator-expr { stmt };
```

Durch Verwendung von *coforall* wird jeder Schleifendurchlauf garantiert in einem Task abgearbeitet und nach Beendigung der Schleife implizit auf Synchronisation gewartet.

Beispiel 4

```
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
```

Um schließlich alle erzeugten Tasks automatisch zu synchronisieren benötigt man das folgende *sync*-Statement.

2.1.5 Dynamische Synchronisation

Syntax

```
sync-stmt:
 sync stmt;
```

Das *sync*-Statement dient zur dynamischen Synchronisation von Tasks. Ein *sync*-Statement synchronisiert dynamisch alle innerhalb des *stmt* erzeugten Tasks im Gegensatz zu *cobegin*.

Beispiel 5

```
sync {
  for i in 1..numConsumers {
   begin consumer(i);
  }

  producer();
}
```

2.1.6 Bedingte Parallelität

Syntax

```
serial-stmt:
 serial expr stmt;
```

Da es Szenarien gibt, in denen zuviel Parallelität schädlich für die Laufzeit eines Programms sein kann, bietet Chapel mit dem *serial*-Statement eine Möglichkeit abhängig von *expr* die Taskerzeugung zu unterbinden. Sofern *expr* zu *true* evaluiert wird, wird innerhalb von *stmt* jegliche Taskerzeugung unterbunden und das gesamte Statement sequentiell ausgeführt.

Beispiel 6

```
def search(i : int) {
  serial i > 8 cobegin {
    search(i*2);
    search(i*2+1);
  }
}
```

2.1.7 Atomic Sections

Um generell Tasks von kritischen Abschnitten auszuschließen sind die bisher vorgestellten Methoden nur bedingt hilfreich. Daher wäre es wünschenswert bestimmte Abschnitte atomar auszuführen. Chapel bietet mit den atomic sections ein Konstrukt um diesen Ausschluss zu implementieren.

Syntax

```
atomic-stmt:
 atomic stmt;
```

Da dieses Feature in Version 0.9 noch nicht fertig implementiert ist, kann sich die Sezifikation dahingehend noch ändern. Aktuell ist eine Implementerung über Software Transactional Memory geplant. Sämtliche Speicheroperationen werden in einem separaten Speicherbereich durchgeführt und nach Abschluss der Transaktion an den Ursprung zurück kopiert. Somit bleiben Speicheroperationen innerhalb der atomic-section bis zum Ende der Transaktion unsichtbar. Derzeit fehlt noch eine Spezifikation der zu verwendenden Atomizität.

schwach Die Atomic-Section ist lediglich vor dem Zugriff durch andere Atomic-Sections geschützt.

stark Die Atomic-Section ist gegen sämtliche Speicherzugriffe geschützt.

Beispiel 7

```
atomic {
  A[i] = A[i] + 1;
}
```

Zur Veranschaulichung der eben vorgestellten Taskparallelität soll folgendes Beispielprogramm dienen, welches eine parallele Variante von QuickSort implementiert:

```
use Random, Time;

var timer: Timer;

config var n: int = 2**15;
config var thresh: int = 1;

var A: [1..n] real;

fillRandom(A);
pqsort(A, thresh);

for i in 2..n do
  if A(i) < A(i-1) then
    halt("A(", i-1, ") == ", A(i-1), " -> A(", i, ") == ", A(i));

writeln("verification success");

def pqsort(arr: [],
           thresh: int,
           low: int = arr.domain.low,
           high: int = arr.domain.high) where arr.rank == 1 {

  if high - low < 8 {
    bubbleSort(arr, low, high);
    return;
```

```
  }

  const pivotVal = findPivot();
  const pivotLoc = partition(pivotVal);

  serial thresh <= 0 do cobegin {
    pqsort(arr, thresh-1, low, pivotLoc-1);
    pqsort(arr, thresh-1, pivotLoc+1, high);
  }

  def findPivot() {
    const mid = low + (high-low+1) / 2;

    if arr(mid) < arr(low) then arr(mid) <=> arr(low);
    if arr(high) < arr(low) then arr(high) <=> arr(low);
    if arr(high) < arr(mid) then arr(high) <=> arr(mid);

    const pivotVal = arr(mid);
    arr(mid) = arr(high-1);
    arr(high-1) = pivotVal;

    return pivotVal;
  }

  def partition(pivotVal) {
    var ilo = low, ihi = high-1;
    while (ilo < ihi) {
      do { ilo += 1; } while arr(ilo) < pivotVal;
      do { ihi -= 1; } while pivotVal < arr(ihi);
      if (ilo < ihi) {
        arr(ilo) <=> arr(ihi);
      }
    }
    arr(high-1) = arr(ilo);
    arr(ilo) = pivotVal;
    return ilo;
  }
}

def bubbleSort(arr: [], low: int, high: int) where arr.rank == 1 {
  for i in low..high do
    for j in low..high-1 do
      if arr(j) > arr(j+1) then
        arr(j) <=> arr(j+1);
}
```

2.2 Datenparallelität

2.2.1 Domänen und Arrays

Viele Algorithmen lassen sich nur unzureichend oder ineffizient mittels Taskparallelismus beschreiben, daher stellt Chapel auch Methoden zu datenparallelem Vorgehen bereit. Um in Chapel datenparallel vorzugehen benötigt man erst einige Definitionen im Zusammenhang mit Arrays in Chapel. Allen voran die Definition einer Domäne.

Definition Domäne
Eine Domäne ist eine Sammlung von Namen (Indices) für Daten mit folgenden Eigenschaften:

- Domänen sind iterierbar.
- Domänen besitzen eine totale Ordnung auf ihren Elementen.
- Domänen können über Locales verteilt werden (Distributions).

Es existieren drei unterschiedliche Domänentypen.

arithmetisch Die gespeicherten Indexwerten bilden kartesische Tupel. Zusätzlich lassen sich arithmetische Domänen noch als 'strided' als auch als 'sparse' deklarieren.

assoziativ Indexwerte sind Haskeys. Damit lassen sich u.a. Hashtabellen implementieren.

undurchsichtig Indexwerte sind anonym. Dieser Typ eignet sich u.a. zur Deklaration von Graphen und Mengen.

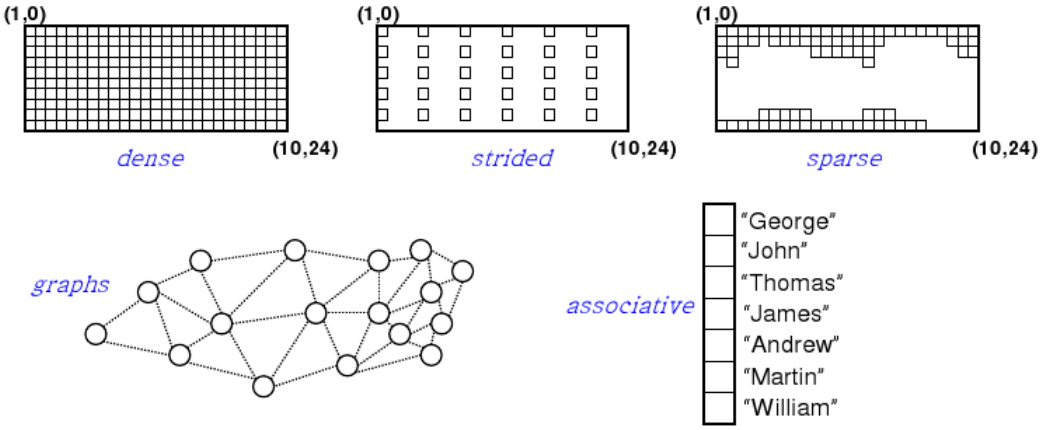


Abbildung 1: Domänentypen in Chapel

Beispiel 8 Domänendeklaration (arithmetisch)

```
var m = 4, n = 8
var D: domain(2) = [1..m, 1..n];
var InnerD: subdomain(D) = [2..m-1, 2..n-1];
var StidedD: subdomain(D) = D by (2, 4);
var SpareseD: subdomain(D) = getIndices();
```

Mit dieser Definition einer Indexmenge benötigt man zusätzlich noch eine Abbildung von Indexwerten auf Daten. Diese Abbildung bildet in Chapel die Definition des Arrays.

Definition Array
Ein Array ist eine Abbildung von Domänen-Indices auf eine Sammlung von Variablen.

Arrayzuweisungen geschehen ausschließlich *by-value* (Im Gegensatz zu Java). Im besonderen ist, unter der Annahme, dass A ein Array-Typ und B vom Typ Array, Range, Domain oder Iterator ist:

```
A = B;
```

äquivalent zu:

```
forall(i,e) in (A.domain, B) do
  A(i) = e;
```

Beispiel 9

```
var A, B: [D] real;
var C: [1..3] int;
var StridedA : [StridedD] real;
```

2.2.2 Schleifen

Zur Iteration über Domänen stehen drei Schleifentypen zur Verfügung. Die einfachste unter ihnen ist die squentielle *for* Schleife. Alle Iterationsschritte laufen im gleichen Task und erzeugen somit keine Parallelität. Die *forall* Schleife, bietet die Möglichkeit zur parallelen Iteration. Ob Iterationen parallel abgewickelt werden hängt von der verwendeten Iterator-Expression ab. Iteriert man über Domänen oder Arrays (hier wird implizit über die Domäne iteriert) wird die Anzahl der erzeugten Tasks von der verwendeten Verteilung bestimmt. Die dritte Möglichkeit zur Iteration ist die bereits vorgestellte *coforall* Schleife. Sie garantiert einen erzeugten Task pro Iterationsschritt.

Beispiel 10

```
for (i, j) in (1..3, 4..6) do
  write(i, " ", j, " ");
```

```
forall i in 1..N do
  a(i) = b(i);
```

```
coforall i in 1..N do
  a(i) = b(i);
```

2.2.3 Promotion, Reduktion, Scan

Promotion Aufbauend auf parallel iterierbaren Schleifen implementiert Chapel nun noch drei Funktionen, um Probleme datenparallel zu lösen.

Mit Hilfe der Promotion lassen sich auf Skalaren definierte Operatoren (=,+,*,min, max, sin, cos, u.a.) auch mit iterierbaren Ausdrücken verwenden, sofern die Elementtypen mit dem Operator kompatibel sind. Chapel verwendet zur Implementierung eine *forall* Schleife, was, abhängig von der Verteilung des Arrays, implizit zu Parallelität führen kann.

Beispiel 11

```
var A, B : [1..5] int;
A = 1;
B = 3;
A = sin(B);
B = 2 * A;
```

Die 1 wird per Promotion allen Arrayfeldern zugewiesen.

Reduktion

Syntax

```
reduce-expr:
 reduce-type reduce iteratable-expr;
```

Neben der Promotion bietet Chapel zusätzlich die Möglichkeit einen Operator wiederholt hierarchisch auf einen iterierbaren Ausdruck anzuwenden, bis er zu einem Skalar reduziert worden ist. Damit lässt sich u.a. die Summe aller Elemente bestimmen. Die Reduktion berücksichtigt, ebenso wie die Promotion, die Verteilung des Arrays und erzeugt dementsprechend Parallelität um die Operation zu beschleunigen.

Beispiel 12

```
var A : [1..5] int;
var tot : int;

A   = 1;
tot = + reduce A;
```

Ergebnis: 5

Scan

Syntax

```
scan-expr:
 scan-type scan iteratable-expr;
```

Die dritte Möglichkeit parallel auf iterierbaren Daten zu arbeiten bietet die Scan Operation. Im Unterschied zu einer Reduktion werden beim Scan die partiellen Reduktionen aller Elemente erstellt. In jeder Iteration wird eine Reduktion auf das aktuelle und alle vorhergehenden Elemente durchgeführt.

Beispiel 13

```
var A, B, C: [1..5] real;
A = 1.1;
B = + scan A; // B = [1.1, 2.2, 3.3, 4.4, 5.5]
B(3) = -B(3); // B = [1.1, 2.2, -3.3, 4.4, 5.5]
C = min scan B; // C = [1.1, 1.1, -3.3, -3.3, -3.3]
```

3 Lokalität und Affinität

Bis zu diesem Punkt fehlt einem als Entwickler noch eine Möglichkeit Datenspeicherung und Tasks gezielt zu steuern. Ein derartiges Feature ist jedoch zur effizienten parallelen Entwicklung mit Chapel zwingend erforderlich. Chapel verwendet dazu das Konzept einer *locale*.

3.1 Lokalität

Definition locale
Ein *locale* ist die strukturelle Einheit um Lokalität in Chapel auszudrücken. Ein *locale* symbolisiert sowohl Rechen- als auch Speicherkapazität.

- Threads innerhalb einer *locale* erhalten annähernd uniform Zugriff auf lokalen Speicher.
- Zugriff auf Speicher innerhalb anderer *locales* ist möglich, erzeugt jedoch, unter Umständen teure Kommunikation.
- *locales* lassen sich ähnlich zu Domänen formen (strided, sparse, dense).

Locales repräsentieren üblicherweise einen Core in einem Multicore Rechner, oder einen Knoten in einem SMP Cluster. In Chapel sind folgende Datenstrukturen im Zusammenhang mit *locales* bereits vordefiniert.

```
config const numLocales : int;
const LocaleSpace : domain(1) = [0..numLocales-1];
const Locales : [LocaleSpace] locale;
```

Die Anzahl der verfügbaren *Locales* sind vom Benutzer zur Ausführungszeit als Parameter zu übergeben.

3.2 Affinität

Mit den *locales* hat man nun eine Möglichkeit, konkret einzelne Daten und Berechnungen an einen bestimmten Ort zu binden. Dies geht sowohl explizit über *on* und *local*, als auch implizit über Verteilungen. Während die Steuerung der *locale*-Zugehörigkeit über *on* und *local* fast schon an Message Passing erinnert, bietet die implizite Steuerung bereits einen hohen Abstraktionsgrad und bleibt dennoch kontrollierbar.

3.2.1 Explizite Steuerung über on/local

Das on-Statement

Syntax

```
on-stmt:
  on expr { stmt };
```

Das *on*-Statement wertet *expr* und führt *stmt* auf der *locale* aus, die durch *expr* bestimmt wird. Dabei ist zu beachten, dass *on* selbst keine Parallelität erzeugt. Dadurch lässt es sich hervorragend mit allen bisher vorgestellten parallelen Statements kombinieren. Man beachte, dass *stmt* sowohl ein Methodenaufruf als auch eine Zuweisung sein kann.

Beispiel 14

```
var A : [LocaleSpace] int;
coforall loc in Locales do on loc {
  A(loc.id) = computation(loc.id);
}
```

Das folgende Beispiel erzeugt auf einer entfernten *locale* eine Objektinstanz.

Beispiel 15

```
class C {
  var x : int;
}

var c : C;
on Locales(1) do c = new C();

writeln((here.id, c.locale.id, c));
```

Das local-Statement

Syntax

```
local-stmt:
  local stmt;
```

Mit Hilfe des *local*-Statements lässt sich explizit Kommunikation unterdrücken. Es wird garantiert, dass innerhalb von *stmt* keine Kommunikation zwischen *locales* stattfindet. Durch diese Garantie können innerhalb des local-Blocks diverse Laufzeitüberprüfungen deaktiviert werden.

Beispiel 16

```
c = Root.child(1);
on c do local {
  traverseTree(c);
}
```

3.2.2 Verteilungen

Um den Überblick über die parallelen Konzepte in Chapel abzuschließen, fehlt an dieser Stelle noch die Möglichkeit implizit eine Affinität für Daten und Prozesse festzulegen.

Definition Verteilung
Eine Verteilung (Distribution) ist eine Datenstruktur, die eine Abbildung von (Domänen)-Indices auf *locales* implementiert.

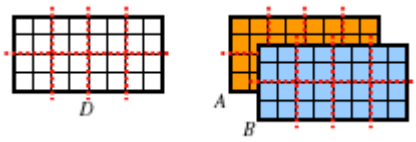
- *Verteilungen* bieten dem Compiler die Möglichkeit Array-Daten auf einzelne *locales* zu verteilen.
- Bei parallelen Iterationen über verteilte Domänen wird implizit festgelegt auf welcher *locale* eine Iteration ausgeführt wird.

Chapel bietet bereits einige vordefinierte Distributionen, wie z.B. die Block-Verteilung. Es ist vorgesehen, benutzerdefinierte Verteilungen anzubieten, welche jedoch derzeit auf Grund von Konsistenzproblemen noch nicht spezifiziert sind.

Beispiel 17

```
var Dist = new Block();
var D : domain(2) dist Dist;
var A,B: [D] int;
```

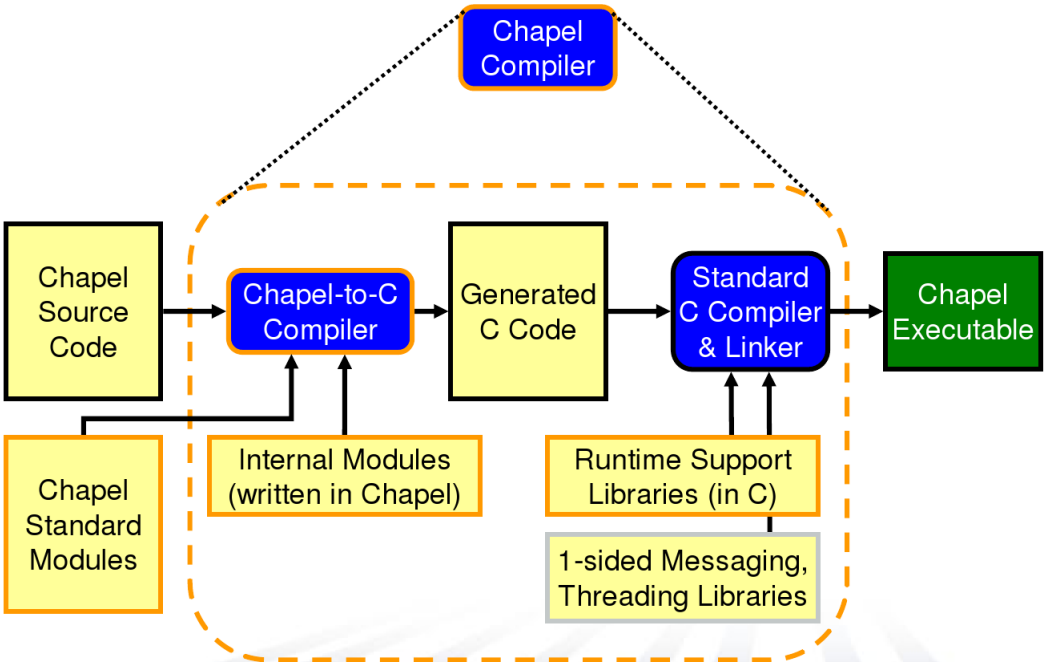
Die Blockverteilung bildet die verfügbaren *locales* in Blöcken auf ein Array ab.



4 Compilerarchitektur

Abschließend soll nun der Chapel Compiler selbst behandelt werden, um ein Verständnis dafür zu erhalten, wie die eben vorgestellten Mechanismen vom Compiler umgesetzt werden. Chapel besitzt einen Source-To-Source Compiler. Der Chapel Code wird in Standard C-Code kompiliert, anschließend wird dann ein (konfigurierbarer) C-Compiler aufgerufen, der dann letztendlich das ausführbare Programm erzeugt.

4.1 Aufbau



Zur Compilezeit werden dem Chapel-Programm zuerst die Standard-Module hinzugefügt (z.B. Time). Anschließend werden dem Programm vom Chapel-To-C Compiler einige interne Module hinzugefügt. Diese implementieren u.a. parallele Operatoren und Funktionen wie z.B. sin, cos, +, *, u.a.. Aus diesem Programm wird nun, z.B. mit gcc C-Code, generiert. Abschließend werden noch benötigte Laufzeitbibliotheken gelinkt und man erhält ein ausführbares Chapel Programm.

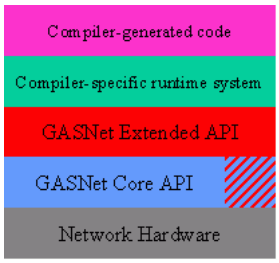
Das Programm wird dynamisch u.a. gegen

- pthreads und
- GASnet

gelinkt. Da alle Tasks in Chapel via Threads implementiert werden, werden alle Tasks über pthreads implementiert. GASnet auf der anderen Seite ist eine Bibliothek die

einen Global Address Space (GAS) implementiert. Durch GASNet ist es Chapel möglich allen *locales* nahezu vollständigen Shared-Memory Zugriff auf allen Variablen zu gewähren. GASNet erzeugt bei Bedarf die nötige Kommunikation über diverse Übertragungsmöglichkeiten wie Infiniband via OpenIB, MPI, Mellanox VAPI, Myrinet GM, UDP, u.a...

Das folgende Bild beschreibt das Laufzeitsystem von GASNet schematisch.



Literatur

[1] Cray. Chapel specification. Technical Report version 0.782, Cray Inc., 2009.

[2] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, pages 291–312, August 2007.

[3] Cray. Chapel tutorial using global hpcc benchmarks: Stream triad, random access, and fft. Technical Report version 1.6, Cray Inc., 2008.

[4] Roxana E. Diaconscu and Hans P. Zima. An approach to data distributions in chapel. *International Journal of High Performance Computing Applications*, pages 313–335, August 2007.

[5] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans.

[6] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The cascade high productivity language. *IEEE Computer Society*, pages 52–60, April 2004.

[7] Chapel: the cascade high productivity language. Präsentation, November 2008.

[8] Introduction to chapel: The cascade high productivity language. Präsentation, November 2008.

[9] Chapel: Productive parallel programming at scale. Videopräsentation, June 2008.