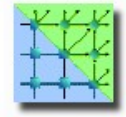
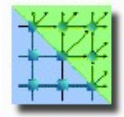
Chapel

Andreas Simbürger

am Lehrstuhl für Programmierung
Seminar: Multicore-Programmierung

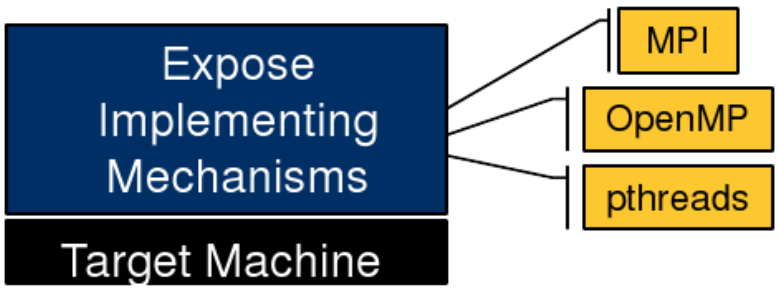
22. Juli 2009



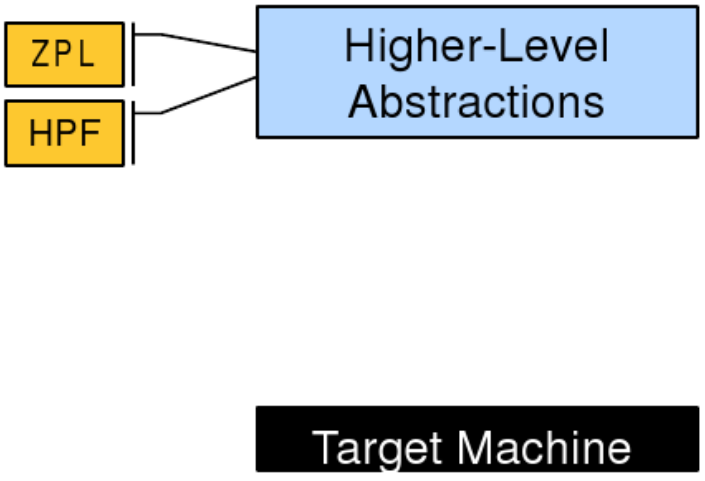


Motivation

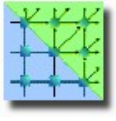
Status quo



"Why is everything so painful?"

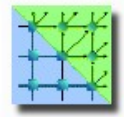


"Why do my hands feel tied?"



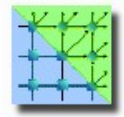
Die Sprache Chapel

- Chapel = Cascade High Productivity Language (CHaPeL)



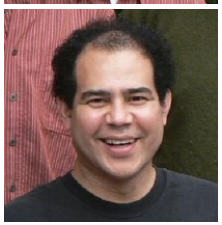
Hintergründe

- Chapel wird entwickelt von Cray im Rahmen des HPCS Projekts von 2002-2010
- 2004 Beginn der Entwicklung an Chapel
- 14. Nov. 2008 - Erster öffentlicher Release (0.8). Seitdem unter der BSD Lizenz
- Aktueller Release: 0.9
- Ab 2010 Übergabe an Open Source Gemeinschaft geplant

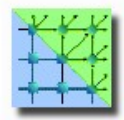


Entwicklerteam

"Know your enemy"

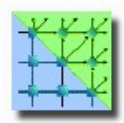


- Brad Chamberlain
  Projektleiter. Früher: ZPL.
- Steve Deitz
  Früher: ZPL.
- Samuel Figueora
- David Iten



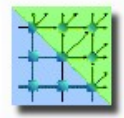
Einflüsse

- ZPL, HPF: Verteilte Arrays, Datenparallelität, Indexmengen
- Cray MTA C/Fortran: Taskparallelität, Synchronization
- CLU, Ruby, Python, C#: Iteratoren
- ML, Scala, Matlab, Perl, Python: Optionale Typangaben
- Java, C#: OOP, Typsicherheit
- C++: Generische Programmierung / Templates
- C, Modula, Ada: Syntax



Sequentielle Sprachkonzepte

- Stil: blockstrukturiert, imperativ
- Syntax: Einflüsse durch C, C++, Java, C#, Perl, u.v.m.
- Objektorientierte Programmierung (optional)
  - Referenz-/ Wertbasierte Objekte
- Typsystem: stark typisiert $\longrightarrow$ Automatische Typinferenz
- Zeiger: Nur Objekt- / Arrayreferenzen; Keine Zeigerarithmetik

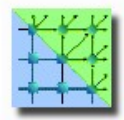


Syntaxhighlights

- Konfigurationskonstanten: **config const** $n = 10$;
- Swap Operator: $<=>$
- Zippered- und Tensor-Iteration
- Named Argument Passing: Argumente werden per Name zugeordnet, nicht per Position
- Parameterübergabe über *in*,*out*,*inout* und *const* regulierbar
- Iteratoren: **yield**
- Generics: **type**

Tool Support!

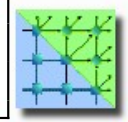
vim und emacs editor modes ;-)



Codebeispiel

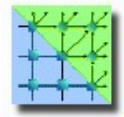
99 bottles of beer

```
config const numBottles = 99;

def main {
  const numVerses = numBottles+1;
  var LyricsSpace: domain(1) = [1..numVerses];
  var Lyrics: [LyricsSpace] string;

  [verse in LyricsSpace] Lyrics(verse) = computeLyric(verse, numVerses);
  writeln(Lyrics);
}

def computeLyric(verseNum, numVerses) {
  var bottleNum = numBottles - (verseNum - 1);
  var nextBottle = (bottleNum + numVerses - 1)%numVerses;
  return "\n" + describeBottles(bottleNum, startOfVerse=true) + " on the wall, "
              + describeBottles(bottleNum) + ".\n" + computeAction(bottleNum)
              + describeBottles(nextBottle) + " on the wall.\n";
}
def describeBottles(bottleNum, startOfVerse = false) {
  var bottleDescription = if (bottleNum) then bottleNum:string
                                         else (if startOfVerse then "N"
                                                               else "n")
                                              + "o more";
  return bottleDescription
       + " bottle" + (if (bottleNum == 1) then "" else "s")
       + " of beer";
}
def computeAction(bottleNum) {
  return if (bottleNum == 0) then "Go to the store and buy some more, "
                             else "Take one down and pass it around, ";
}
```



Grundlage für Parallelität in Chapel ist ein ideales SPMD Modell.

- Homogene Knoten (Rechenleistung, Architektur)
- Alle verfügbaren Datenspeicher sind Shared-Memory.



Das **begin** statement:

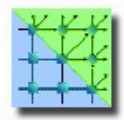
Syntax

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

- Erzeugt einen parallelen Task (stmt)

Beispiel

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



Der **sync** Datentyp

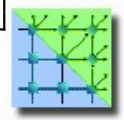
Syntax

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

- Eine **sync**-Variable kann zusätzlich zu ihrem aktuellen Wert noch zwei Zustände annehmen (voll/leer).
- Lesezugriffe blockieren, bis die Variable als 'voll' markiert ist.
  Nach dem Lesezugriff wird die Variable auf 'leer' gesetzt.
- Schreibzugriffe blockieren, bis die Variable erneut leer ist.
- Voll/Leer Verhalten ist über diverse Lese-/ Schreibmethoden anpassbar.

Beispiel

```
var lock: sync bool;

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



Das **cobegin** Statement.

Syntax

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

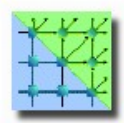
- Es wird für jedes Statement aus der stmt-list nicht-blockierend ein Task generiert.
- Am Ende des **cobegin** Blocks wird blockierend gewartet. (Impliziter Join)
- Syntaktischer Zucker: Kann durch **begin**-Statements in Verbindung mit **sync** Variablen ersetzt werden.

Beispiel

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

⇔

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



Das **coforall** Statement.

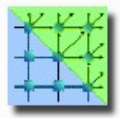
Syntax

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

- **coforall** generiert für jeden Schleifendurchlauf einen Task.
- Am Ende der **coforall** Schleife wird blockierend gewartet. (Impliziter Join)

Beispiel

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



Das **sync** Statement.

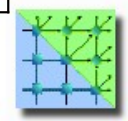
Syntax

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

- Das **stmt** wird ausgeführt.
- Am Ende des **sync** Blocks wird blockierend auf alle im Block aufgetretenen **begin** Statements gewartet (Impliziter Join)
- Auch innerhalb von Tasks neu erzeugte Tasks werden berücksichtigt (im Gegensatz zu **cobegin**).

Beispiel

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



Das **serial** Statement

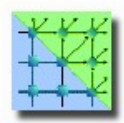
Syntax

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

- Der Ausdruck **expr** wird evaluiert.
- (expr == true) => Jegliche Parallelität innerhalb von **stmt** wird unterdrückt.
- (expr == false) => **stmt** wird ausgeführt.

Beispiel

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



Das **atomic** Statement

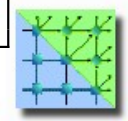
Syntax

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

- **stmt** wird ausgeführt als wäre es eine elementare Operation.
- Schreibzugriffe innerhalb von **stmt** sind bis zum Ende des Blocks außerhalb unsichtbar.
- Implementierung über Software Transactional Memory geplant
- **Derzeit nocht nicht implementiert.**
  Besonders die Wahl zwischen **starker** (Atomar zum gesamten Programm) und **schwacher** (Atomar zu allen atomaren Blöcken im Programm) Atomizität ist noch zu treffen.

Beispiel

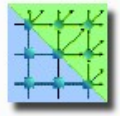
```
atomic {
  A[i] = A[i] + 1;
}
```



Codebeispiel

Parallel Quicksort

- siehe quicksort.chpl



Definition (Domäne)

Eine Domäne ist eine Sammlung von Namen (Indices) für Daten.

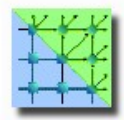
- Domänen sind iterierbar.
- Domänen besitzen eine totale Ordnung auf ihren Elementen.
- Domänen können über Locales verteilt werden (Distributions).

In Chapel unterscheidet man drei Arten von Domänen:

arithmetisch: Indexwerte bilden kartesische Tupel. Arithmetische Domänen sind rechteckig und können sowohl 'strided' als auch 'sparse' sein.

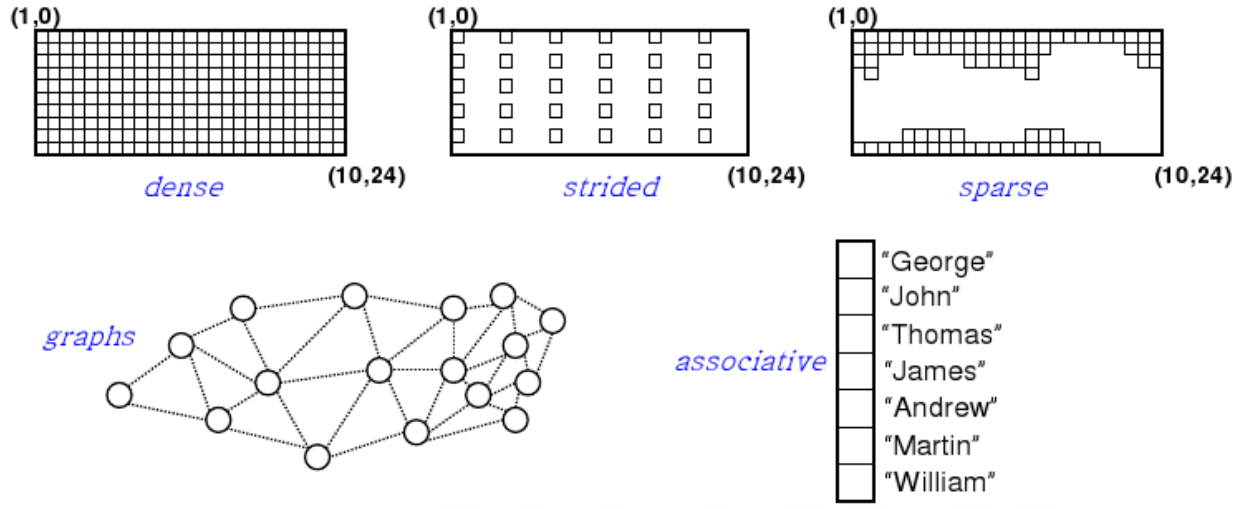
assoziativ: Indexwerte sind Hashkeys (Hashtabellen).

undurchsichtig: Indexwerte sind anonym (Mengen, Graphen).



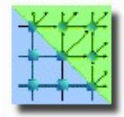
Domänentypen

Beispiel Domänentypen



Beispiel

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



Definition (Array)

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

Beispiel

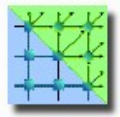
```
var A, B: [D] real;
var C: [1..3] int;
var StridedA : [StridedD] real;
```

- Arrayzuweisung geschieht immer “by-value”.
- Vorausgesetzt A ist ein Array-Typ und B ist vom Typ Array, Range, Domäne oder Iterator, dann ist:

```
A = B;
```

$\Leftrightarrow$

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



Iteration über Domänen

Chapel bietet 3 Arten von Iterationen über Domänen.

for Sequentielle for-Schleife: Alle Iterationen laufen im aufrufenden Task.

forall Abhängig von der verwendeten Iterator-Expression werden Tasks zur Abarbeitung der Iterationen erzeugt.

- Bei Iteration über Domänen bzw. Arrays, wird die Anzahl der möglichen Tasks durch die Distribution bestimmt.
- Ansonsten muss der Programmierer explizit Taskparallelismus im Schleifenrumpf verwenden.

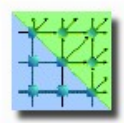
coforall Es wird garantiert ein Task pro Iteration erzeugt.

Beispiele

```
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);
```



Promotion, Scans, Reduktionen

Promotion Skalare Operatoren / Funktionen lassen sich automatisch parallel auf Array-Elemente anwenden.

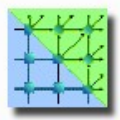
Reduktion Ein Operator wird auf alle Elemente eines iterierbaren Ausdrucks angewandt und zu einem Skalar aggregiert.

Scans Ein Scan erstellt die partiellen Reduktionen über einen iterierbaren Ausdruck.

Beispiele

```
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]
```

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

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



Definition (locale)

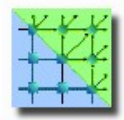
Ein *locale* ist die strukturelle Einheit um Lokalität in Chapel auszudrücken. Ein *locale* symbolisiert sowohl Rechen- als auch Speicherkapazität.

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

Beispiel

Mögliche Kandidaten für eine *locale* sind ein Core in einem Multicore Rechner, oder ein Knoten in einem SMP Cluster.
Vordefiniert stehen zur Verfügung:

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



Steuerung der Affinität

on/local

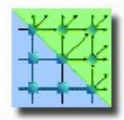
Syntax

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

- Das Statement *stmt* wird auf dem *locale*, das durch *expr* bestimmt wird, ausgeführt.
- Beachte: *on* erzeugt keine Parallelität.

Beispiel

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



Steuerung der Affinität

on/local

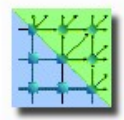
Syntax

```
local_stmt :
  local stmt;
```

- Es wird garantiert keine Kommunikation erzeugt.
- Schaltet einige Laufzeitüberprüfungen ab.

Beispiel

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



Verteilungen

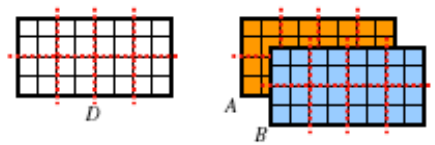
Definition (Verteilung)

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

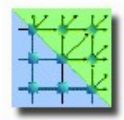
- *Verteilungen* bieten dem Compiler die Möglichkeit Array-Daten auf einzelne *locales* zu verteilen.
- Bei parallelen Iterationen über verteilte Domänen wird implizit festgelegt, auf welcher *locale* eine Iteration ausgeführt wird.

Beispiel

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



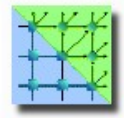
- Derzeit sind lediglich einige vordefinierte Verteilungen verfügbar u.a. Block.
- Benutzerdefinierte Verteilungen werden implementiert, sind aber derzeit noch nicht spezifiziert.



Codebeispiel

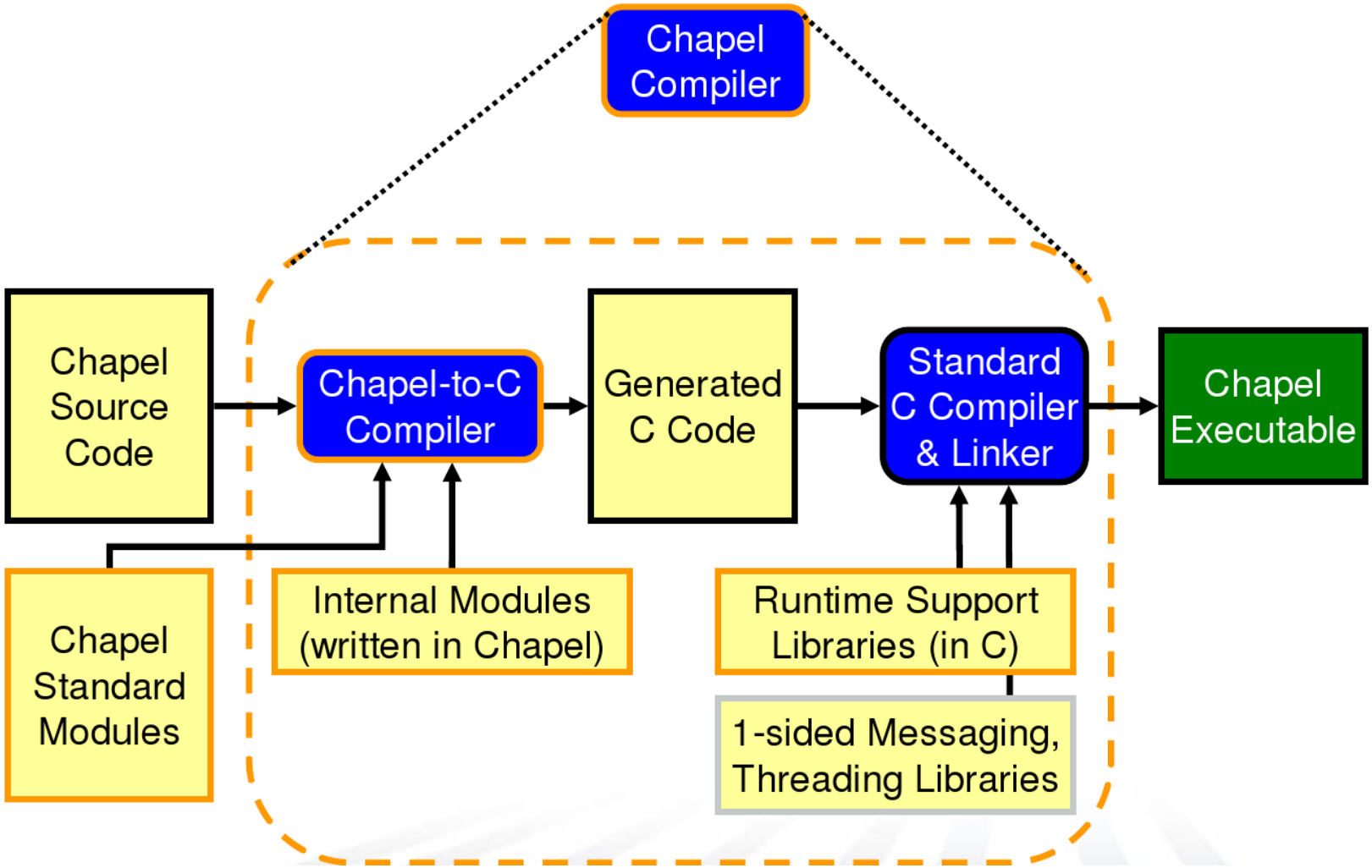
Block Verteilung, 2D

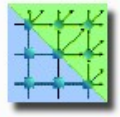
- siehe block2D.chpl



Compilerarchitektur

Übersicht



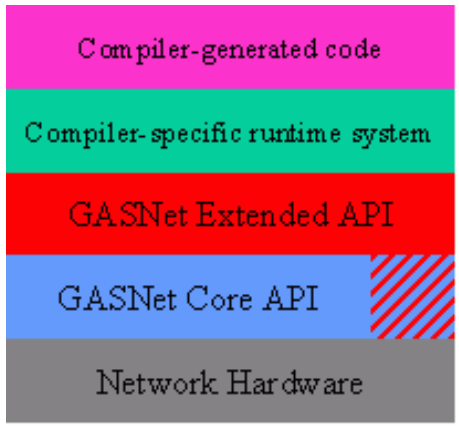


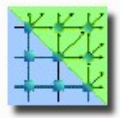
Laufzeitsystem

GASnet, pthreads, u.a.

Chapel Programme werden automatisch gegen verschiedene Laufzeitbibliotheken gelinkt.

- Parallele Tasks werden per default via pthreads realisiert.
  (Abschaltbar, dann jedoch keine Parallelität mehr möglich)
- Zugriff auf Variablen ist von jeder **locale** aus möglich.
  Konzept: Partitioned Global Address Space (PGAS).
  Hier: GASnet.

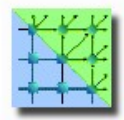




Zusammenfassung

Chapel - Cascade High Productivity Language

- Stark getypte imperative Programmiersprache mit optionaler Objektorientierung
- Sowohl Taskparallelität via begin, cobegin, coforall
- Als auch Datenparallelität via Promotion, Reduktion, Scans, forall, Domänen und Verteilungen
- Synchronisation via sync-Variablen und -Statements



Vielen Dank für die Aufmerksamkeit

Weiterführende Informationen: http://chapel.cray.com
Fragen?